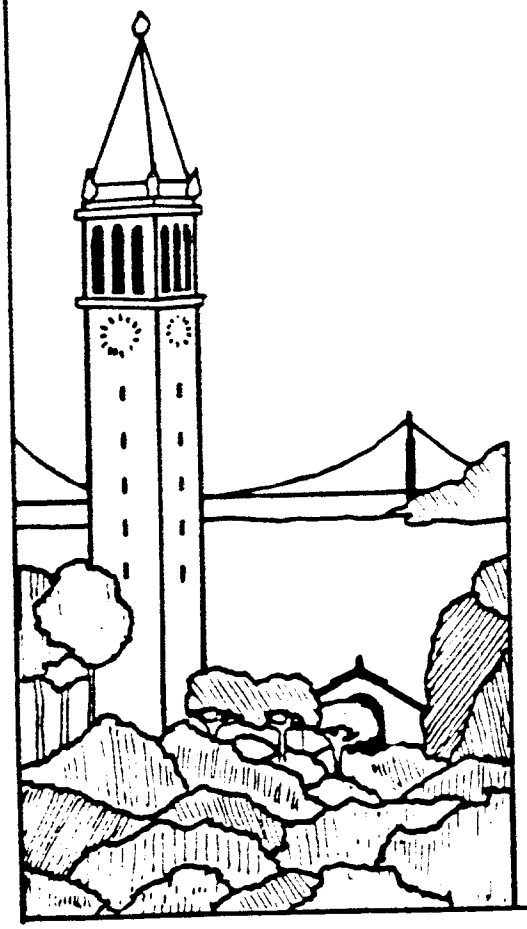
Graphic Presentation of Data Structures in the DBX Debugger

David B. Baskerville



Report No. UCB/CSD 86/260

October 1985

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
OCT 1985		**00-00-1985 to 00-00-1985**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Graphic Presentation of Data Structures in the DBX Debugger	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Debugging is a task that requires access to extensive information about a program and its execution state. The more effectively this information can be presented to the user by a debugger, the better tool the debugger becomes. Graphics is a meaningful and effective way of presenting a program's data structures. This paper describes the design and implementation of an extension to a standard debugger. The extension presents data structures graphically and enables a user to control the format and extent of information provided. The extension to the UNIX debugger DBX runs on a Sun Workstation using multiple windows, a bitmap display, and a mouse pointing device.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**56**	

Graphic Presentation of Data Structures in the DBX Debugger

David B. Baskerville

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, California 94720

ABSTRACT

Debugging is a task that requires access to extensive information about a program and its execution state. The more effectively this information can be presented to the user by a debugger, the better tool the debugger becomes. Graphics is a meaningful and effective way of presenting a program's data structures.

This paper describes the design and implementation of an extension to a standard debugger. The extension presents data structures graphically and enables a user to control the format and extent of information provided. The extension to the UNIX† debugger DBX runs on a Sun Workstation‡ using multiple windows, a bitmap display, and a mouse pointing device.

July 5, 1985

Sponsored by the Defense Advance Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089. David Baskerville was supported by an NSF Graduate Fellowship, grant number RCD-84-50040.

† UNIX is a Trademark of Bell Laboratories

‡ Sun Workstation and *SunWindows* are Trademarks of Sun Microsystems, Inc.

Table of Contents

Graphic Presentation of Data Structures in the DBX Debugger

David B. Baskerville

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, California 94720

1. Overview

This paper describes the design and implementation of an extension to a standard debugger. The extension presents data structures graphically and enables a user to control the format and extent of information provided. The extension to the UNIX debugger DBX [UNIX 1984] runs on a Sun Workstation under the *SunWindows* window manager [Sun 1983]. The system is successful in presenting even complex data structures in an effective manner with good performance. A user may move and modify data structures on the screen and may control the amount of information presented about records, arrays, or linked structures.

Sections 2 through 5 of this paper discuss desirable presentation features of a debugger and delineate the objectives of the graphical debugger. Section 6 illustrates the graphical presentation of data structures accomplished by the extended DBX. Section 7 describes the commands for graphic display. Section 8 describes in some depth the implementation of the graphic extension. The concluding sections relate performance results and ideas for further work.

2. The Debugging Problem

Data structures are fundamental to computer programs. A computer program can often be characterized by the data structures it creates and the operations it performs upon them.

In designing a program, or attempting to understand a program, a visual representation of the program's data structures is often essential. Introductory programming courses teach students to think about data structures in *box* form with pointers represented by *arrows* from one box to another. Experience has proven this *box-and-arrow* form to be a worthwhile way of conceptualizing data structures. Advanced programmers think about data structures in the same way, visualizing *box-and-arrow* models of linked lists, binary trees, and symbol tables.

In debugging a program, the user has access to a potentially vast amount of information about the program and its state. The debugger should be able to organize and present this information in a way that is useful and natural to the user. In particular, the most common presentation task of a debugger is to show the value of a variable, which may be a simple value or a data structure. Therefore, it is incumbent on the debugger to be capable of

presenting such data structures in the same form in which users think of them. Linked structures should be presented as boxes joined by arrows from one box to another, and nested structures should appear visually as nested boxes. During interactive debugging, if data structures are presented as users naturally think about them, errors can be spotted more quickly and an understanding of the program's execution can be achieved more readily.

Traditionally, debuggers have presented information as text, making no attempt to present data structures in a *box-and-arrow* format. Attempts to do so on a low resolution ascii terminal would prove futile. The biggest drawback of text-based debuggers is the way in which *pointers* are presented. These debuggers display the value of a pointer as a numerical address of the structure pointed to is given. Yet it is the object pointed to, not its address, that is usually of interest. The user must follow pointers, printing out components of a linked structure individually, to reconstruct the state of a data structure on his scratch paper. If a textual debugger were to present all the objects pointed to, the flood of information would quickly scroll off the screen.

3. The Solution

The advent of high-resolution bit-mapped displays affords the possibility for debuggers to present data structures as the user envisions them. Since the implementation is no longer limited to text, a debugger can draw nested boxes and interconnecting arrows to represent a data structure, ameliorating the debugging process.

Presenting data structures in the familiar *box-and-arrow* form alone, however, is not enough. It is vital that the user be allowed to control the format and extent of information displayed about a data structure. A user may conceive of binary trees very differently from linked lists, even though the underlying data structures are essentially the same. The user should be able to specify how certain data structures are formatted on the screen so that the presentation matches his conception. He should also be able to move the data structures once they have appeared on the screen. This capability enables the user to alter the way he views a data structure or to improve a layout that becomes inappropriate as data structures change dynamically.

Presenting all the information held in a complex data structure may be too much detail. It is important to allow the user to tailor a data structure's presentation to show only those parts which he would like to focus upon. Such control can be provided by allowing the user to close a certain substructure, to suppress a pointer and its linked structure, or to elide parts of an array.

4. Objectives of the Graphical Debugger

The UNIX debugger DBX is a symbolic, source language debugger, supporting C, Pascal, Fortran, and assembly language. Breakpoints may be set at source code line numbers, at the beginning of procedures, or upon a condition which the user specifies. After a breakpoint, the execution of the program can be continued in single step mode, procedures may be called, etc.

A variable's value is printed by entering its name symbolically, using its source code name. If the variable is a record, its value is printed using one line per field. Pointer values are printed as numerical addresses.

The extension to DBX described in this report, which presents data structures graphically and allows user-tailorability, will be referred to as GDBX, for Graphical-DBX.

To achieve the previously described goals of improved debugger presentation, the graphics extension project set the following objectives:

- A variable's value should be presented in *box-and-arrow* format. Structures and nested structures should present information and dependencies clearly.
- A variable's value should be updated after each *execution step*, with changes easily identified. (An *execution step* means any further execution of the program being debugged. This may result from such DBX commands as STEP, NEXT, CONT, CALL, or RUN.)
- Users should be able to control the amount of information presented about a data structure and its layout on the screen, even before beginning to debug a program.
- Users should be able to change the presentation of a data structure by opening or closing fields of a record, by moving structures or arrows, or by scrolling arrays.
- The data structures should be displayed on a *virtual* screen, larger than the physical screen. The virtual screen should be *scrollable*, allowing the visible portion of the screen to be moved up/down or left/right.

5. Related Work

[Model 79] and [Myers 80] give a history of debuggers and defend the use of an *analogical* display for data structures. An *analogical* display is one that makes use of bit-mapped graphics to present objects in the form of boxes, arrows, or icons, creating *analogies* to the physical world.

The work most closely related to GDBX was done by Myers at Xerox PARC. INCENSE [Myers 80] was perhaps the first system to present graphical displays of data structures in a standard compiled language using a bit-mapped display.

INCENSE displays data structures in *box-and-arrow* format, with boxes linked by curved arrows. INCENSE includes a mechanism that allows users to define, by a Mesa program, the presentation format of a record structure.

The placement of linked structures is accomplished using what Myers calls a *layout* mechanism. The *layout* mechanism gives all objects, at the time of their creation, a specified area in which to display themselves. Parts of a linked data structure that are pointed to must find a space for themselves within this area. Thus, components further down the pointer chain shrink themselves to fit into the designated area. Beyond a certain minimum size these structures do not present themselves.

GDBX extends the idea of graphical presentation of data structures in the spirit of INCENSE. Areas in which GDBX improves upon INCENSE solutions are the following:

- Integration into a standard debugger: GDBX is an extension to a standard debugger, whereas INCENSE is loosely connected to the Mesa debugger: "Although presently INCENSE does not have an acceptable front end, it should increase the effectiveness of any debugger into which it might be integrated." [Myers 80, p.1]
- Two-dimensional space allocation: GDBX employs a two-dimensional space allocation algorithm (albeit simple) to find places for linked structures on the screen. INCENSE shied away from this solution and developed the *layout* mechanism. The GDBX solution is more general and can be extended to incorporate a specialized placement routine.

GDBX introduces the following ideas to graphical debugger displays:

- Dynamic user control over data structure presentation: OPEN/CLOSEing of record fields and array elements, moving structures and arrows, scrolling arrays
- Scroll bars and a virtual screen
- Construction and presentation control: SHOW/UNSHOW, PTR_DEPTH, ARRAY_BEGIN/ARRAY_SIZE, SUPPRESS/UNSUPPRESS

6. Illustrations of Graphic Presentation

This section contains screen dumps from a Sun Workstation taken during sessions with GDBX. These screens show how data structures are presented and show operations performed upon the data structures. They illustrate how GDBX meets the objectives outlined above.

(Please refer to section 7 and the MAN page in the appendix for a more detailed discussion of the GDBX commands and mouse operations used in making these screens.)

There are 10 sub-sections illustrating the following displays and operations:

6.1 Display of Data Structures in C and Pascal

6.2 Construction/Modification of Linked Data Structures

6.3 ACROSS/DOWN Layout Specification

6.4 OPEN/CLOSEing of a Record Field

6.5 Structure Movement

6.6 Arrow Movement

6.7 Array/Pointer Ellipsis

6.6 Array Scrolling

6.9 Array Suppression

6.10 Scroll Bars and Font Changes

6.1. Display of Data Structures in C and Pascal

The following screens illustrate how data structures are presented in C and Pascal. The presentation of a data structure is identical in the two languages, with the exception of type names and scalar types of Pascal.

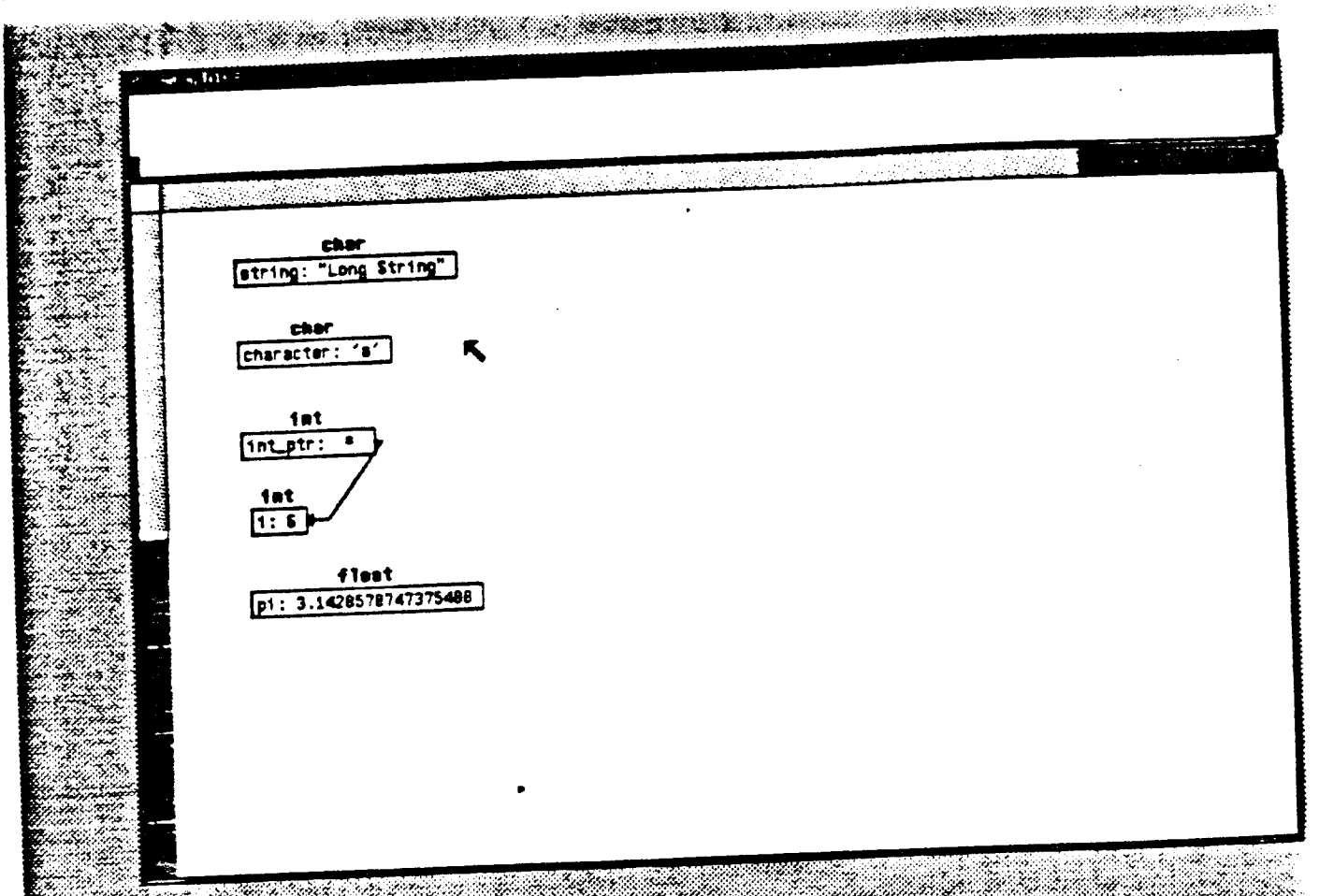
6.1.1 This screen shows the paradigmatic box form in which simple variables are presented. The variable's name is shown on the left side of the box and its value on the right. A variable of pointer type is represented by a value of '*' and an arrow emanating from the right side of the box. Here, C variables of type integer, character, real, string, and pointer are presented.

6.1.2 This screen presents Pascal variables of type integer, character, boolean, real, scalar type, pointer, and record. A record is presented using nested boxes. The record itself is an enclosing box. The fields of a record are shown as boxes stacked vertically within the outer box.

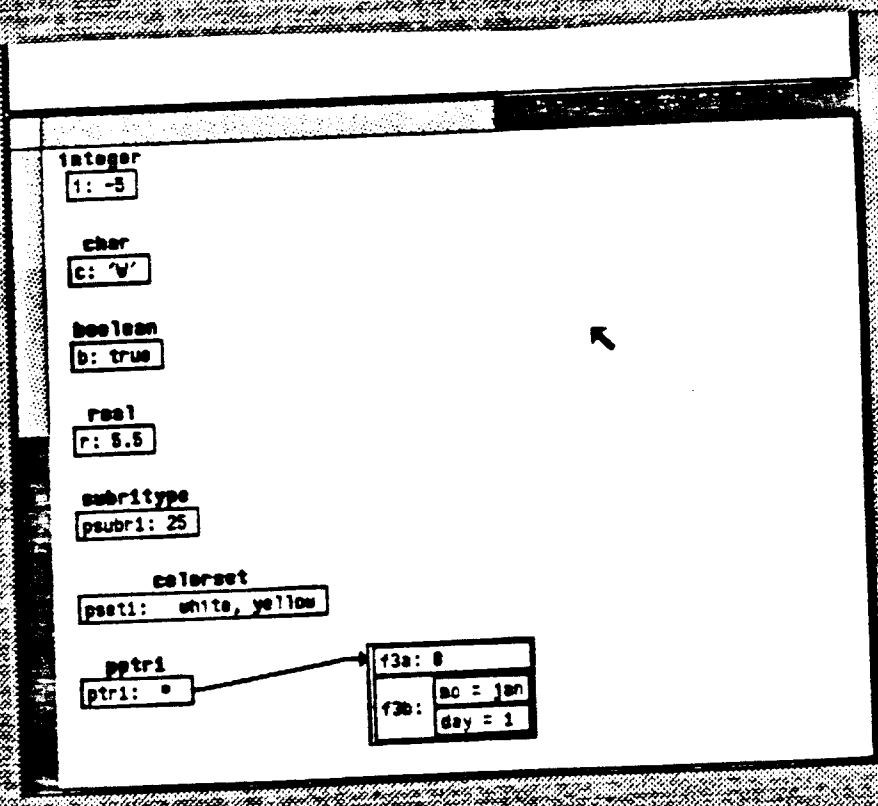
6.1.3 This screen illustrates an array of strings, an array of pointers, an array of integers, and a two-dimensional array. The presentation of arrays is similar to records. An array is shown as an enclosing box. Each element of the array is displayed in a separate box stacked vertically within the box representing the array. The element number is shown in brackets after the array name.

6.1.4 This screen shows an array of pointers, an array of integers, a record, a nested record, and a recursive structure.

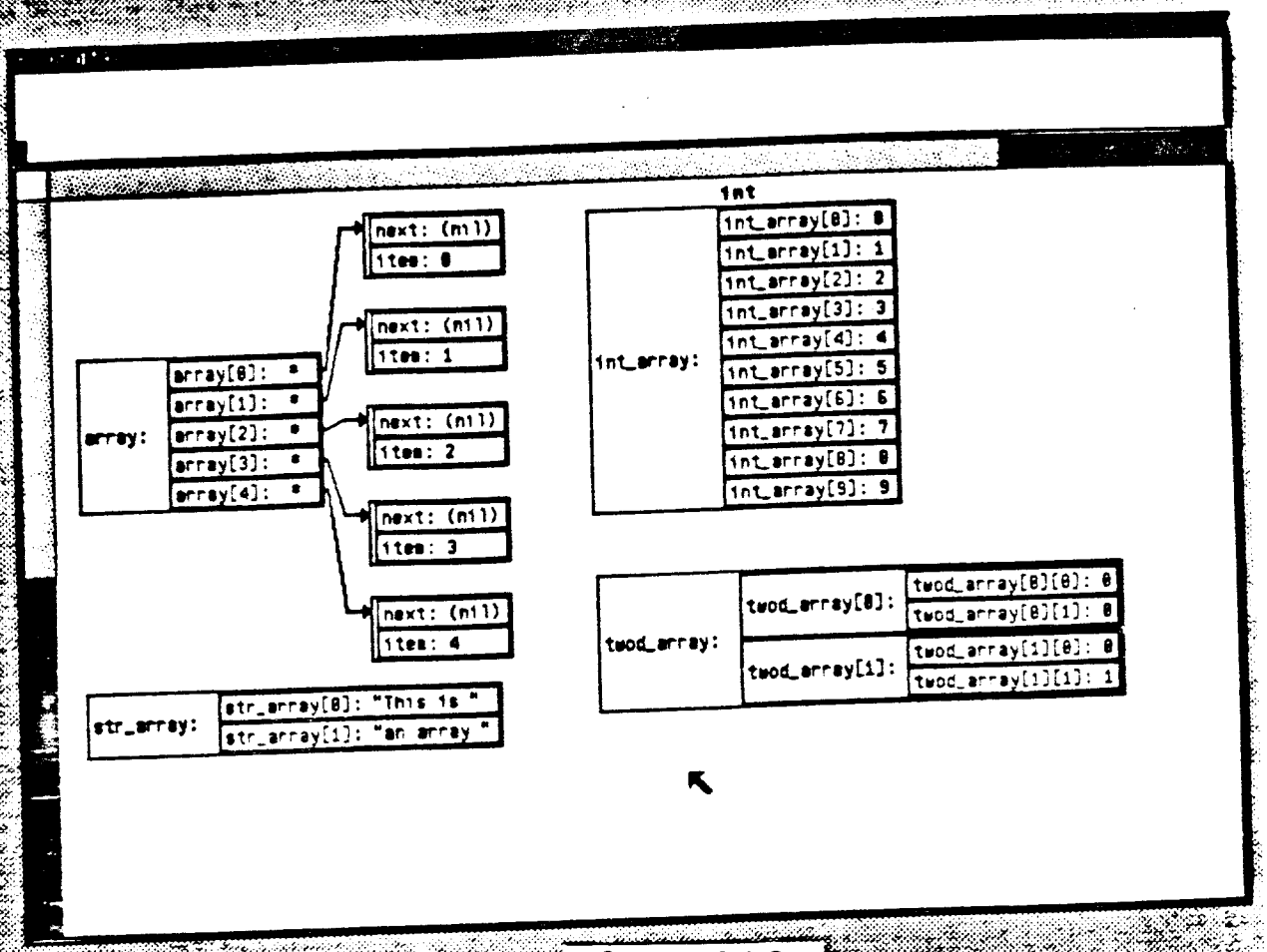
6.1.5 This screen shows a three-dimensional array of scalar literals in Pascal.



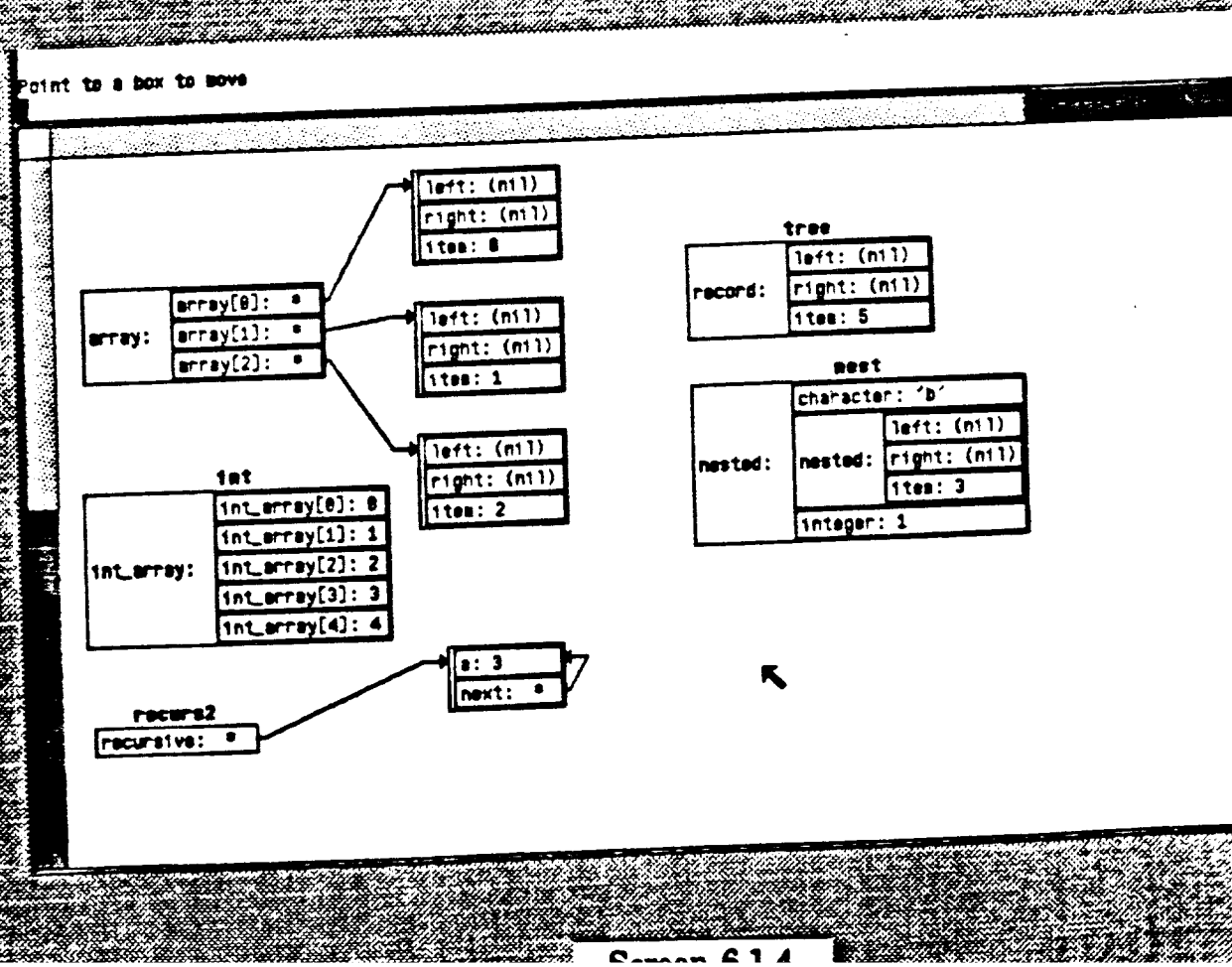
Screen 6.1.1



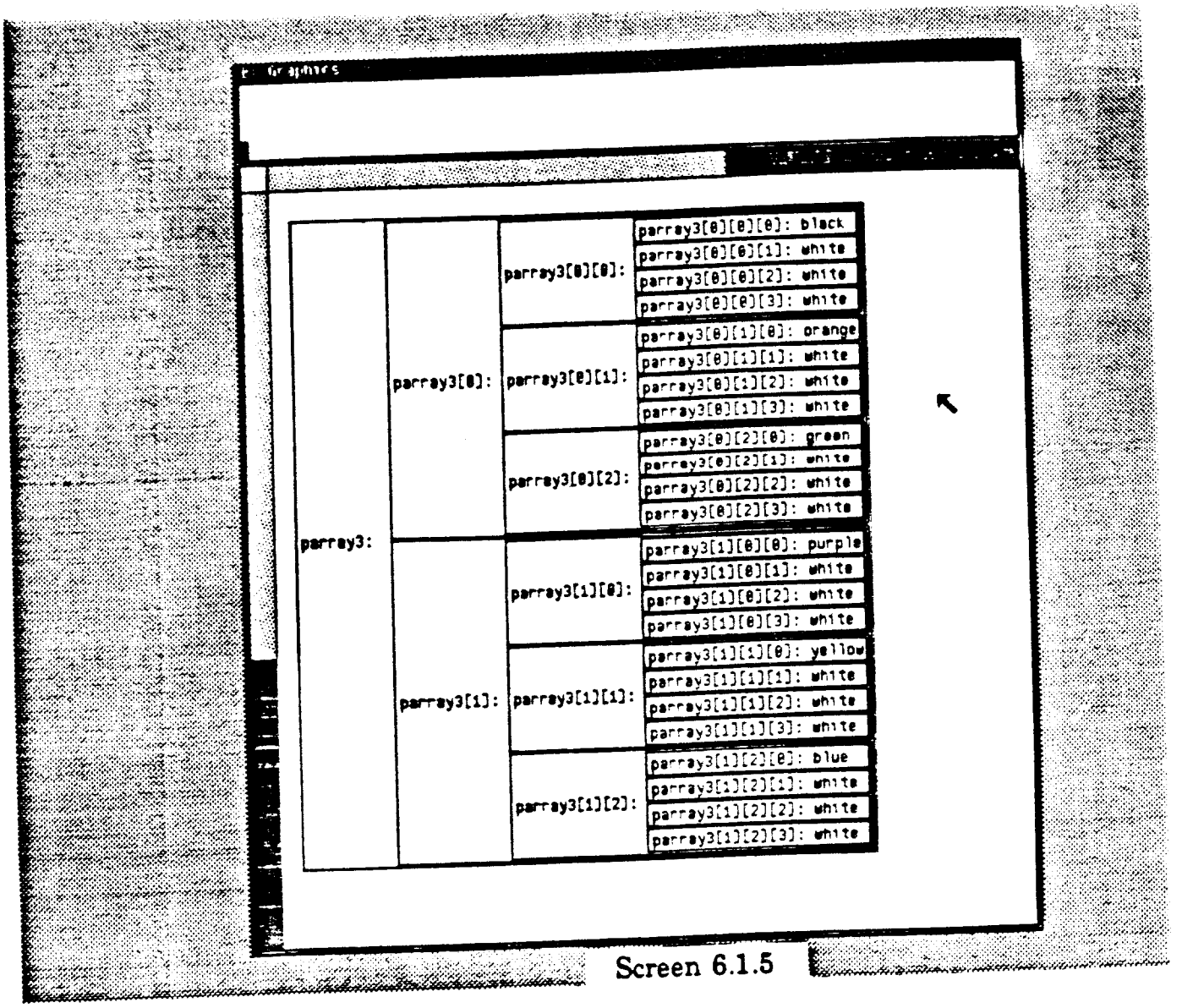
Screen 6.1.2



Screen 6.1.3



Screen 6.1.4


parray3:
parray3[0]:
parray3[0][0]:
parray3[0][0][0]: black
parray3[0][0][1]: white
parray3[0][0][2]: white
parray3[0][0][3]: white
parray3[0][1]:
parray3[0][1][0]: orange
parray3[0][1][1]: white
parray3[0][1][2]: white
parray3[0][1][3]: white
parray3[0][2]:
parray3[0][2][0]: green
parray3[0][2][1]: white
parray3[0][2][2]: white
parray3[0][2][3]: white
parray3[1]:
parray3[1][0]:
parray3[1][0][0]: purple
parray3[1][0][1]: white
parray3[1][0][2]: white
parray3[1][0][3]: white
parray3[1][1]:
parray3[1][1][0]: yellow
parray3[1][1][1]: white
parray3[1][1][2]: white
parray3[1][1][3]: white
parray3[1][2]:
parray3[1][2][0]: blue
parray3[1][2][1]: white
parray3[1][2][2]: white
parray3[1][2][3]: white

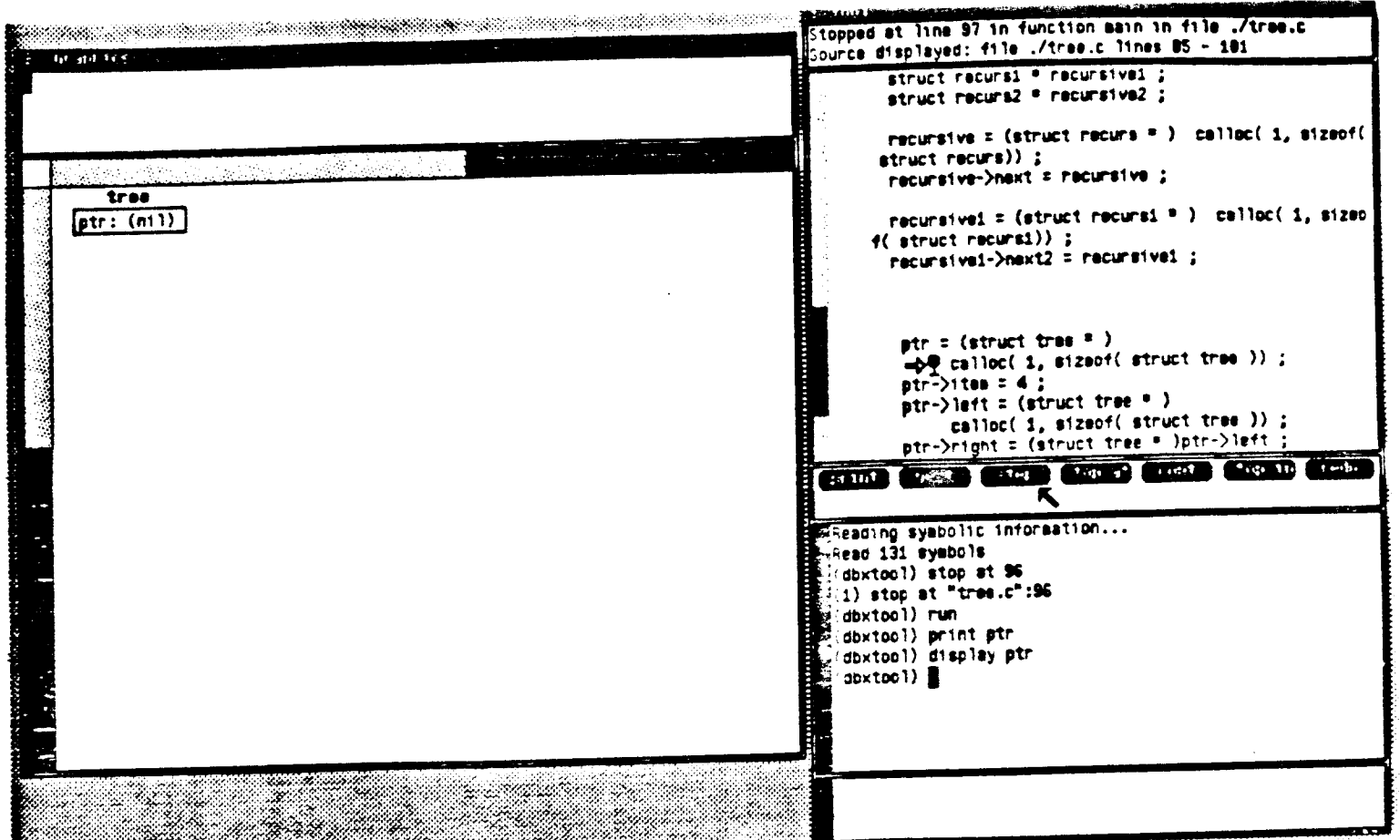
Screen 6.1.5

6.2. Construction/Modification of a Linked Data Structure

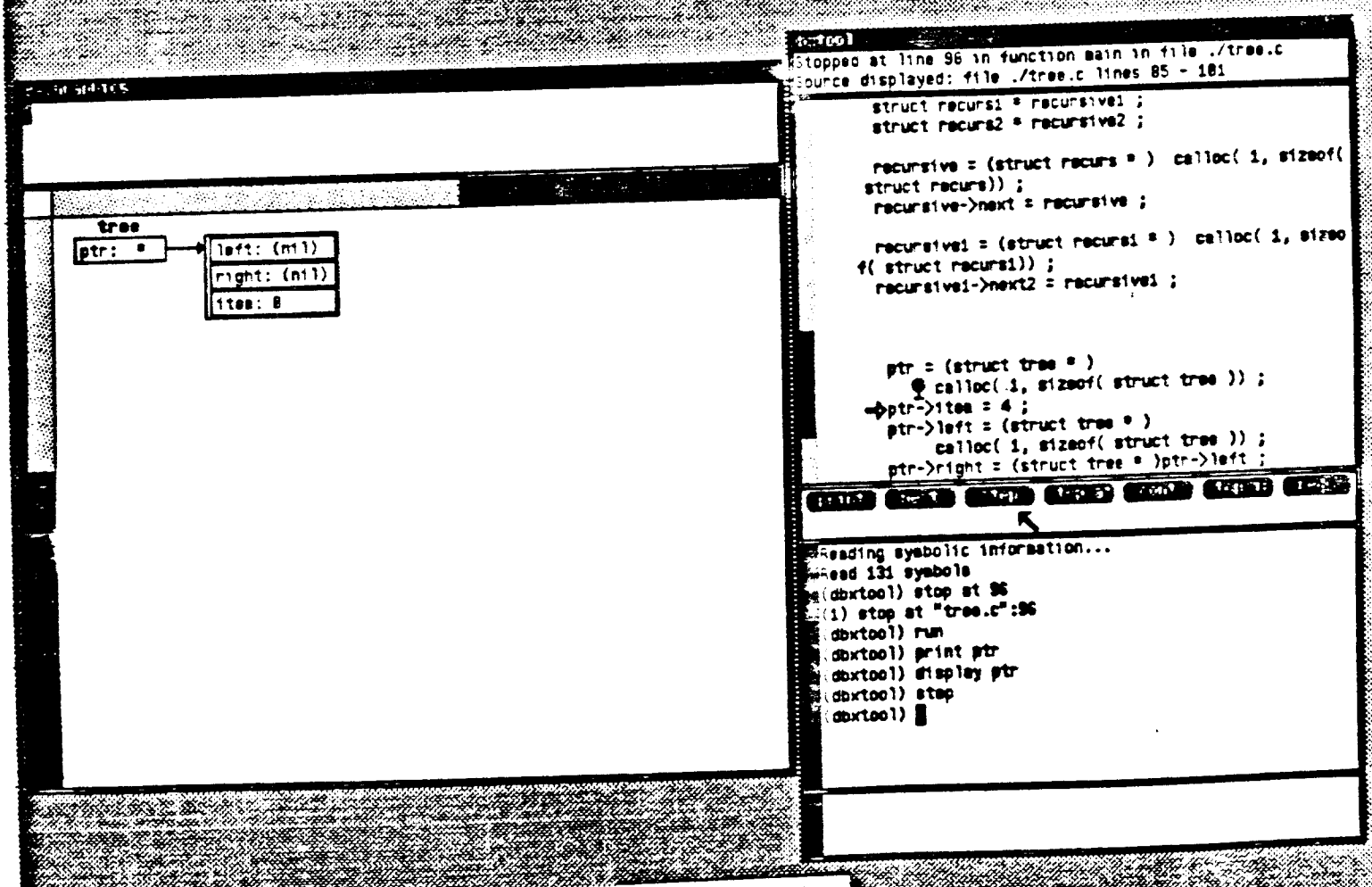
The following screens illustrate a program executed one step at a time using DBXTOOL and GDBX. The program constructs and modifies a linked data structure.

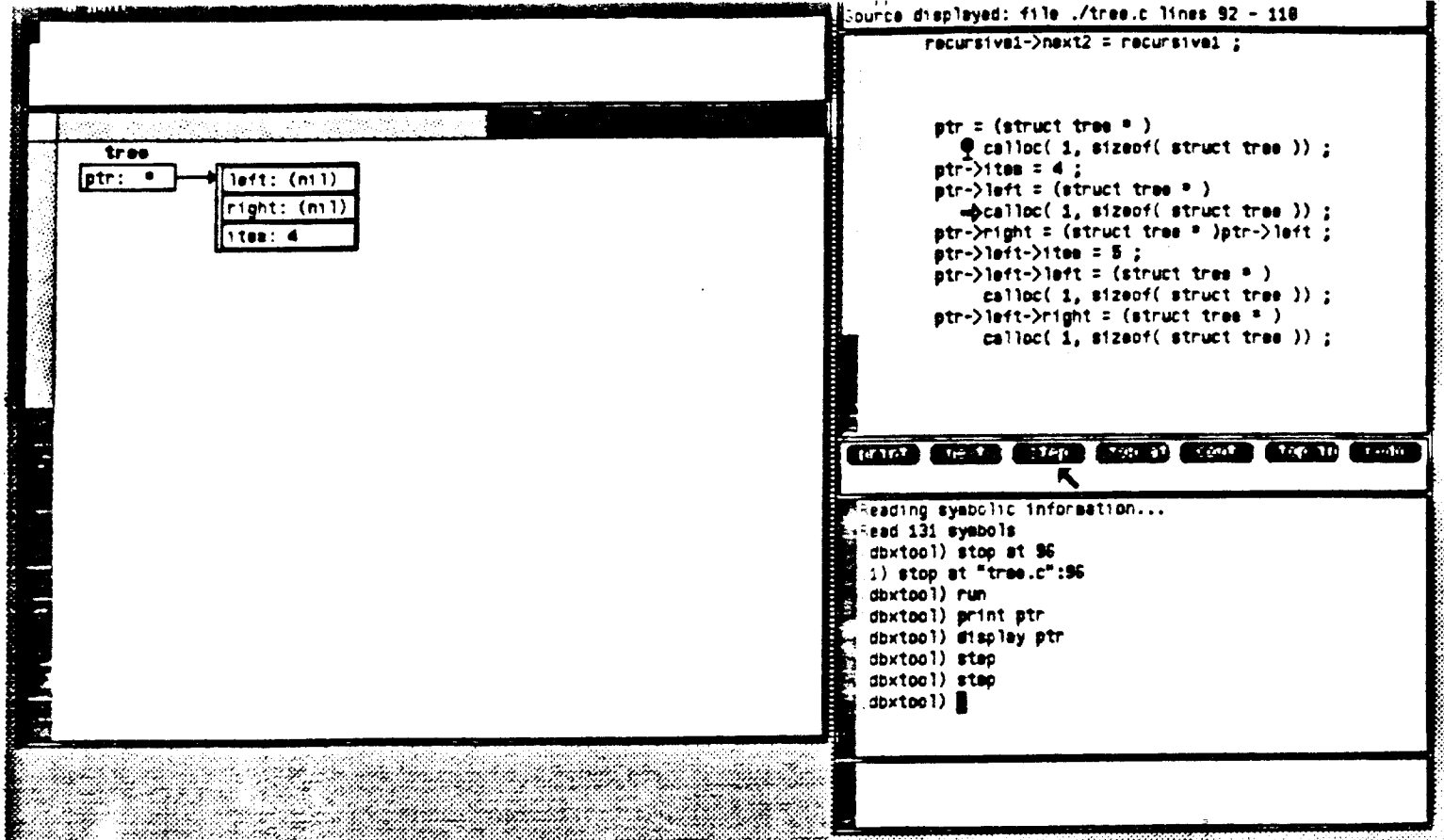
DBXTOOL appears on the right with five subwindows. (See *Integration with DBXTOOL* below.) A breakpoint that has been set is indicated by a stopsign. The current execution stop is shown by a double-shafted arrow. The program is executed one step at a time by clicking the *step* button in the third subwindow.

As each step is executed, the data structure is updated in the GDBX window on the left. This series of screens illustrates how the progress of a program can be seen visually.

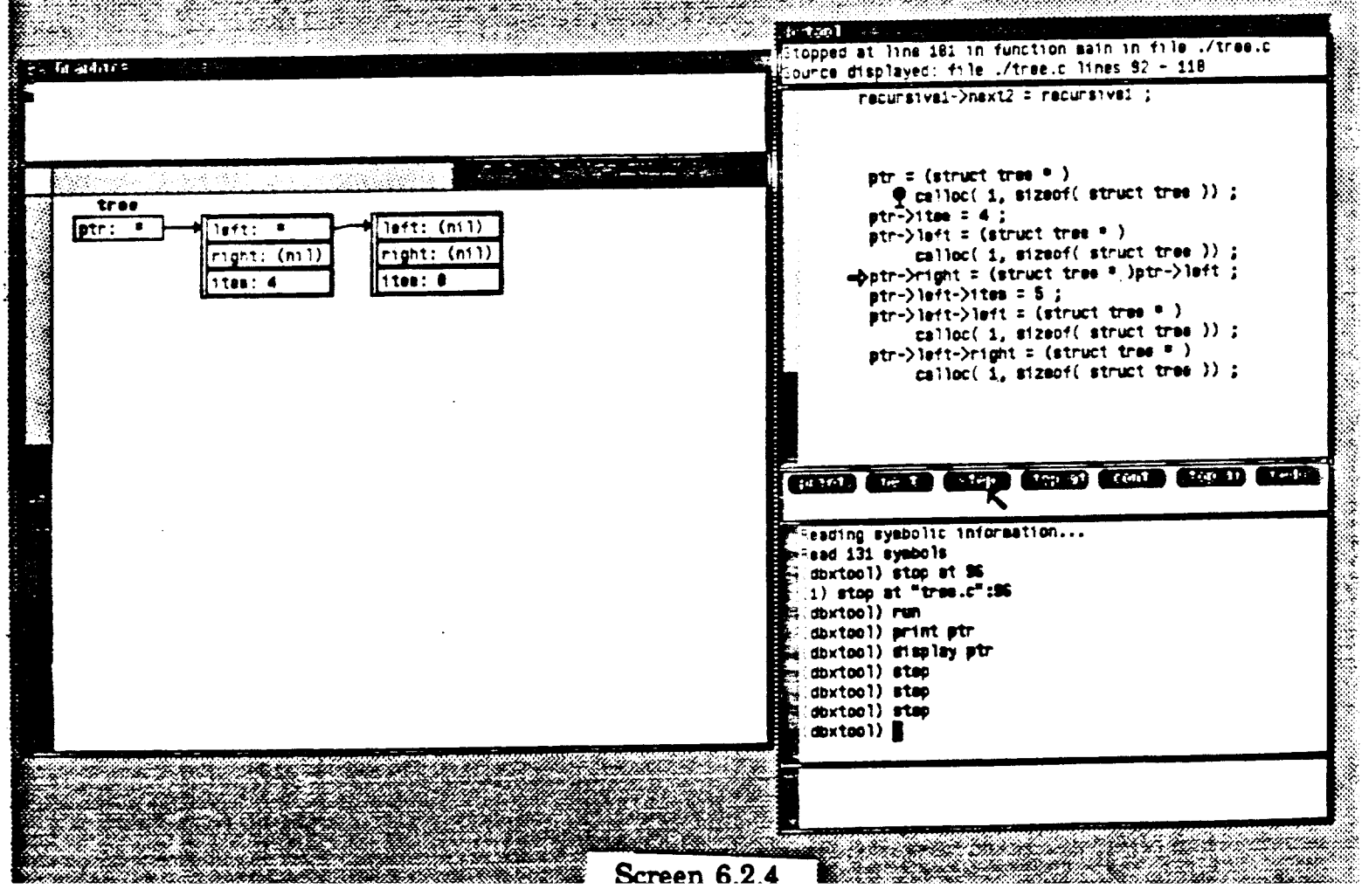

tree
ptr: (nil)
Stopped at line 97 in function main in file ./tree.c
Source displayed: file ./tree.c lines 85 - 101
struct recurs1 * recursive1 ;
struct recurs2 * recursive2 ;
recursive = (struct recurs *) calloc(1, sizeof(
struct recurs)) ;
recursive->next = recursive ;
recursive1 = (struct recurs1 *) calloc(1, sizeo
f(struct recurs1)) ;
recursive1->next2 = recursive1 ;
ptr = (struct tree *)
calloc(1, sizeof(struct tree)) ;
ptr->item = 4 ;
ptr->left = (struct tree *)
calloc(1, sizeof(struct tree)) ;
ptr->right = (struct tree *)ptr->left ;
Reading symbolic information...
Read 131 symbols
(dbxtool) stop at 96
(1) stop at "tree.c":96
(dbxtool) run
(dbxtool) print ptr
(dbxtool) display ptr
(dbxtool)

Screen 6.2.1

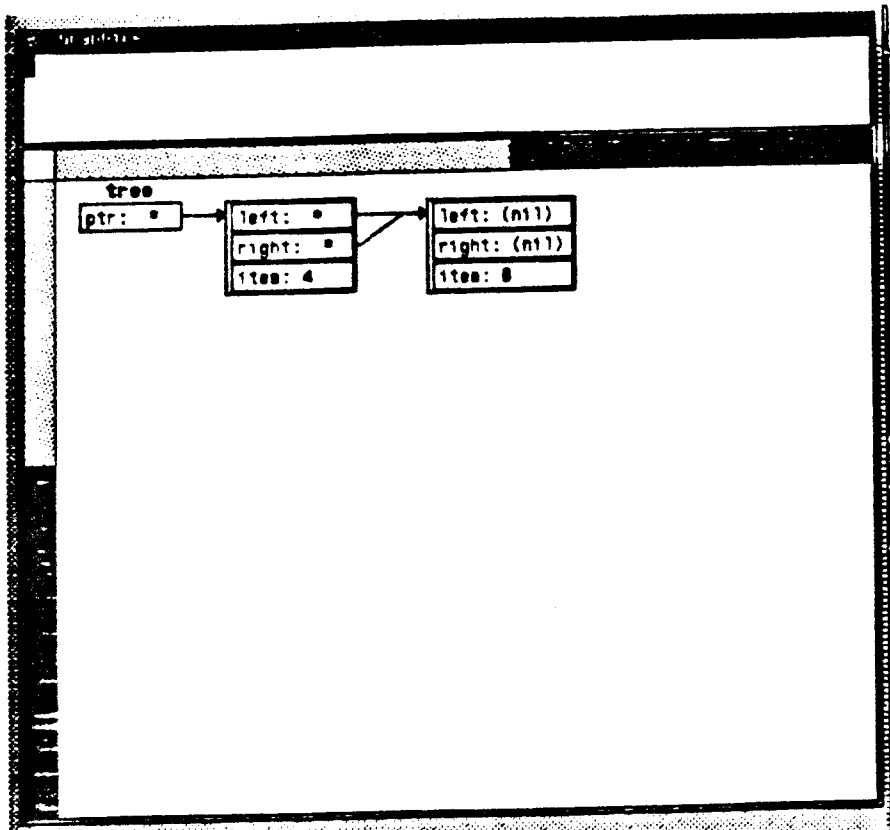

tree
ptr: *
left: (nil)
right: (nil)
item: 8
Stopped at line 96 in function main in file ./tree.c
Source displayed: file ./tree.c lines 85 - 101
struct recurs1 * recursive1 ;
struct recurs2 * recursive2 ;
recursive = (struct recurs *) calloc(1, sizeof(
struct recurs)) ;
recursive->next = recursive ;
recursive1 = (struct recurs1 *) calloc(1, sizeo
f(struct recurs1)) ;
recursive1->next2 = recursive1 ;
ptr = (struct tree *)
calloc(1, sizeof(struct tree)) ;
ptr->item = 4 ;
ptr->left = (struct tree *)
calloc(1, sizeof(struct tree)) ;
ptr->right = (struct tree *)ptr->left ;
Reading symbolic information...
Read 131 symbols
(dbxtool) stop at 96
(1) stop at "tree.c":96
(dbxtool) run
(dbxtool) print ptr
(dbxtool) display ptr
(dbxtool) step
(dbxtool)



Screen 6.2.3



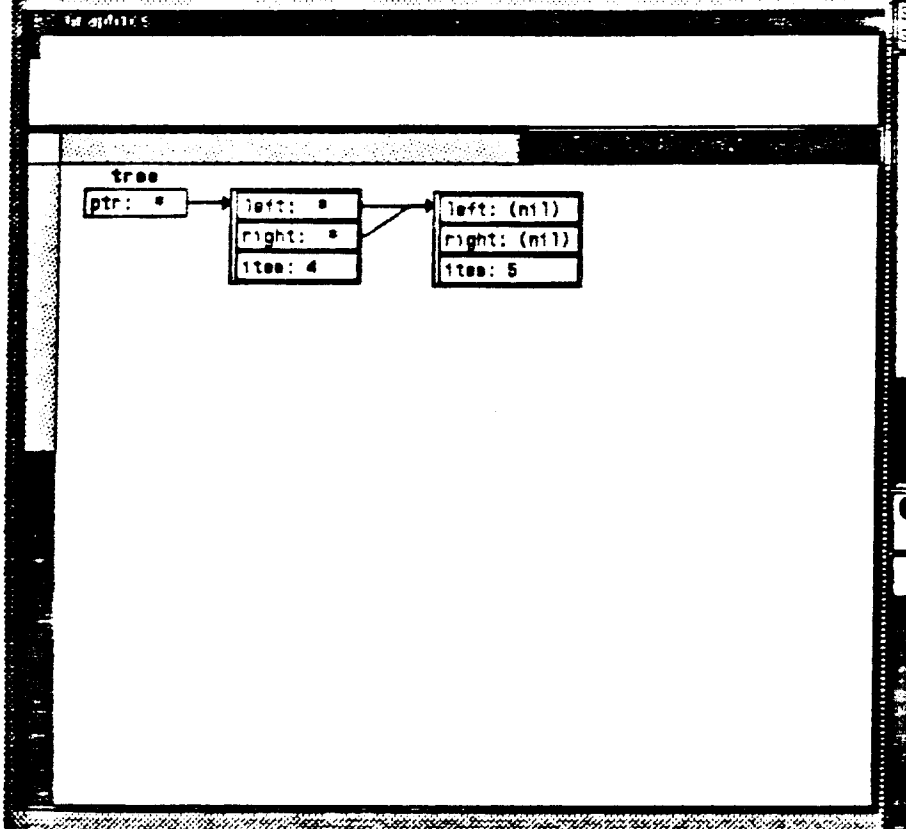
Screen 6.2.4




```
Stopped at line 102 in function main in file ./tree.c
Source displayed: file ./tree.c lines 92 - 118
        recursive1->next2 = recursive1 ;


        ptr = (struct tree * )
          calloc( 1, sizeof( struct tree )) ;
        ptr->item = 4 ;
        ptr->left = (struct tree * )
             calloc( 1, sizeof( struct tree )) ;
        ptr->right = (struct tree * )ptr->left ;
        ptr->left->item = 5 ;
        ptr->left->left = (struct tree * )
             calloc( 1, sizeof( struct tree )) ;
        ptr->left->right = (struct tree * )
             calloc( 1, sizeof( struct tree )) ;
```

```
Reading symbolic information...
Read 131 symbols
(dbxtool) stop at 96
1) stop at "tree.c":96
(dbxtool) run
(dbxtool) print ptr
(dbxtool) display ptr
(dbxtool) step
(dbxtool) step
(dbxtool) step
(dbxtool) step
(dbxtool)
```

Screen 6.2.5



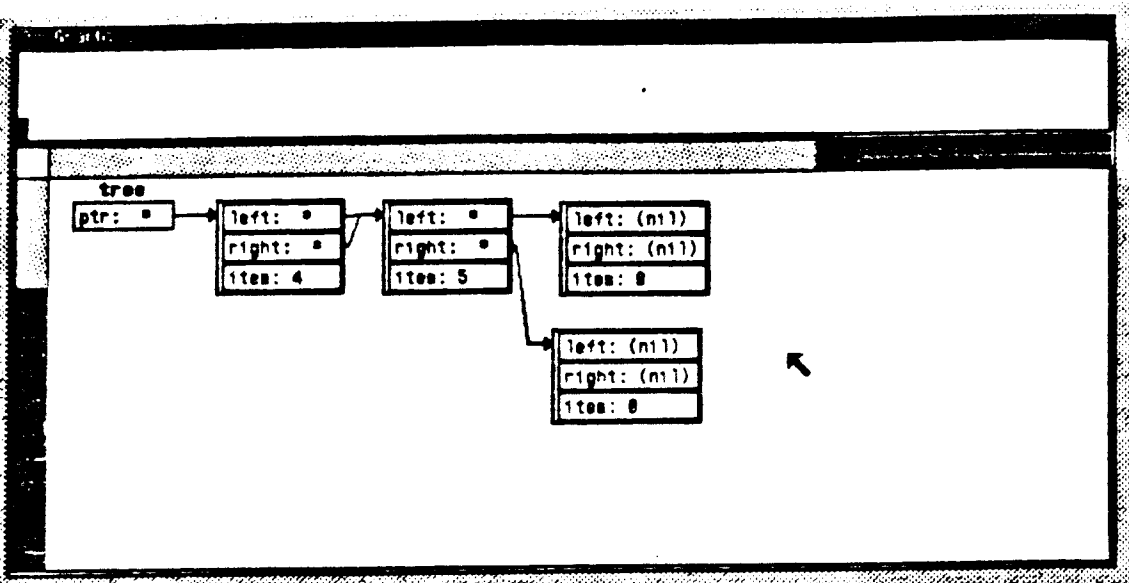

```
Stopped at line 104 in function main in file ./tree.c
Source displayed: file ./tree.c lines 92 - 118
        recursive1->next2 = recursive1 ;


        ptr = (struct tree * )
          calloc( 1, sizeof( struct tree )) ;
        ptr->item = 4 ;
        ptr->left = (struct tree * )
             calloc( 1, sizeof( struct tree )) ;
        ptr->right = (struct tree * )ptr->left ;
        ptr->left->item = 5 ;
        ptr->left->left = (struct tree * )
          =>calloc( 1, sizeof( struct tree )) ;
        ptr->left->right = (struct tree * )
             calloc( 1, sizeof( struct tree )) ;
```

```
Read 131 symbols
(dbxtool) stop at 96
1) stop at "tree.c":96
(dbxtool) run
(dbxtool) print ptr
(dbxtool) display ptr
(dbxtool) step
(dbxtool) step
(dbxtool) step
(dbxtool) step
(dbxtool) step
(dbxtool)
```

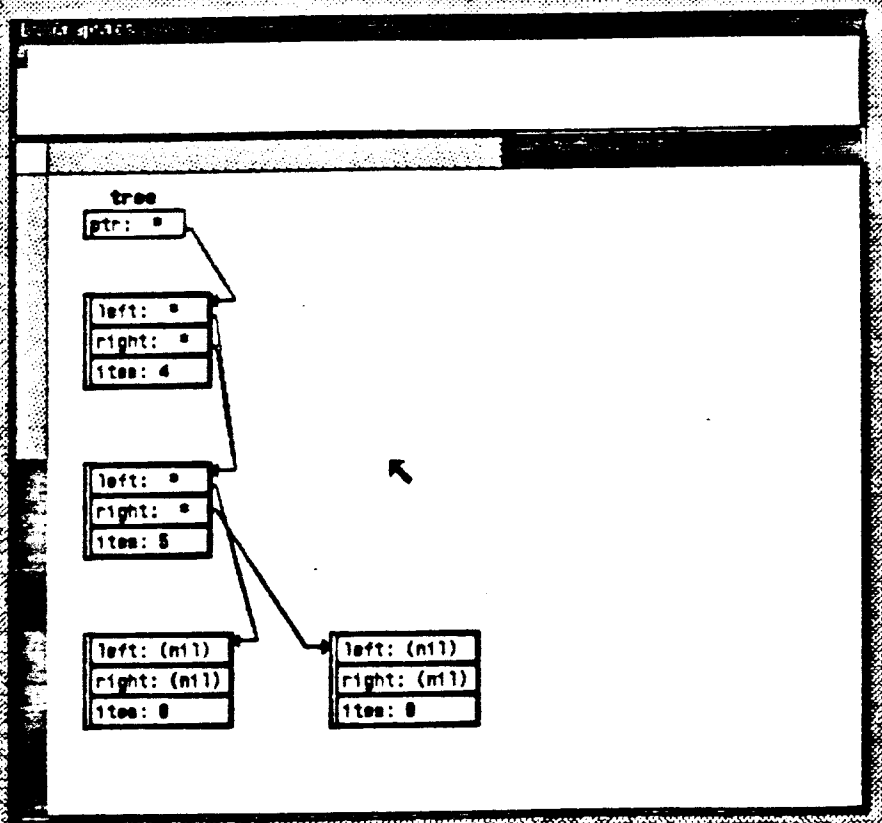
Screen 6.2.6

6.3. ACROSS/DOWN Layout Specification

The following screens show the linked structure constructed in the previous program displayed first ACROSS and then DOWN the screen.



Screen 6.3.1



Screen 6.3.2

6.4. OPEN/CLOSEing of a Record Field

The following screens illustrate an operation on the linked structure constructed above. A field of a record is first selected, then CLOSED or OPENED.

6.4.1 The linked data structure is displayed ACROSS the screen.

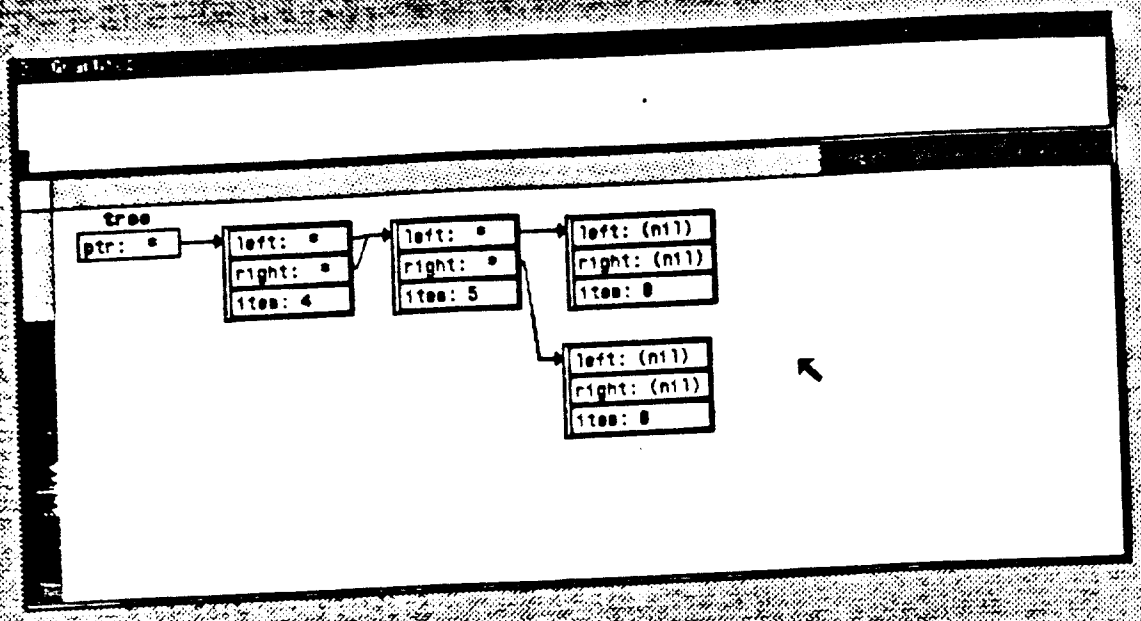
6.4.2 Clicking the left mouse button while pointing at a box selects and highlights the top-level structure. Here, the second box is selected.

6.4.3 Clicking the left mouse button within a highlighted record structure selects a field within the structure. Here, the field named *right* is selected.

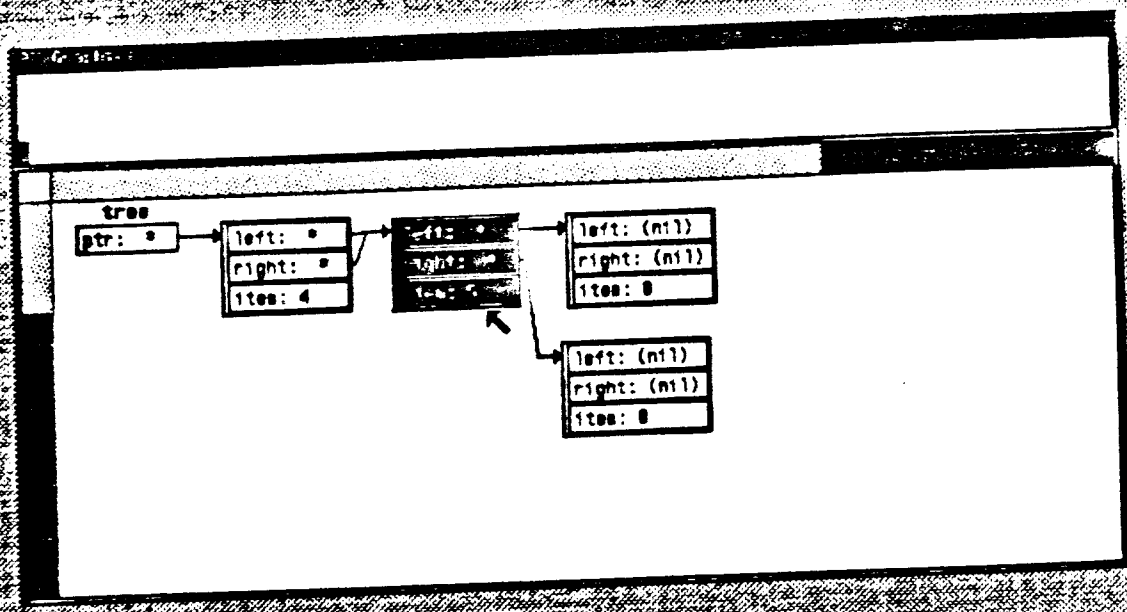
6.4.4 Depressing the right mouse button summons a menu of operations that may be applied to the selected record field. Here, the operation *Close Selection* is chosen. As illustrated, the field named *right* is *closed* and the pointer and structure emanating from it are erased. A closed pointer is indicated by the value '***'.

6.4.5 The closed record field *right* is once again selected.

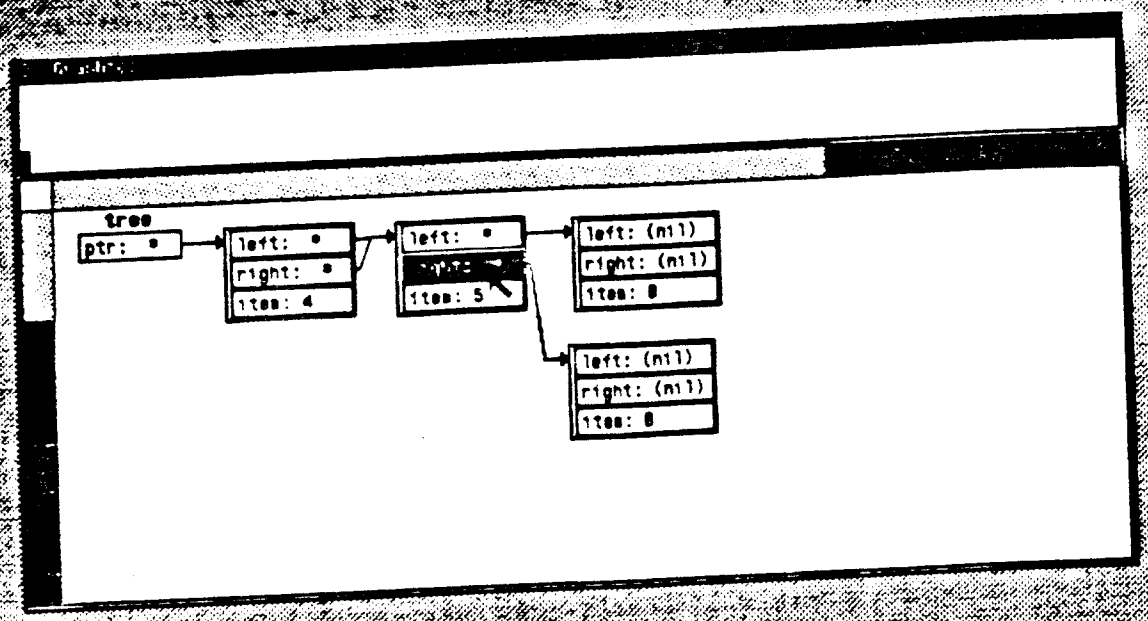
6.4.6 A menu is called and the operation *Open Selection* is applied. This re-opens the *right* field, and the structure to which this field points is once again displayed.



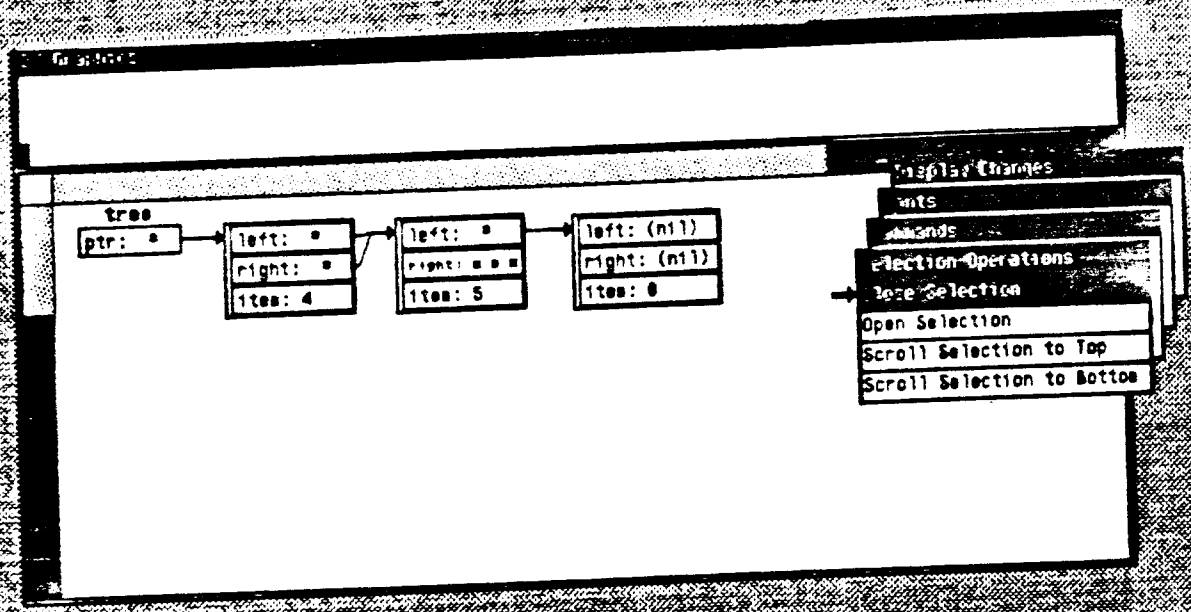
Screen 6.4.1



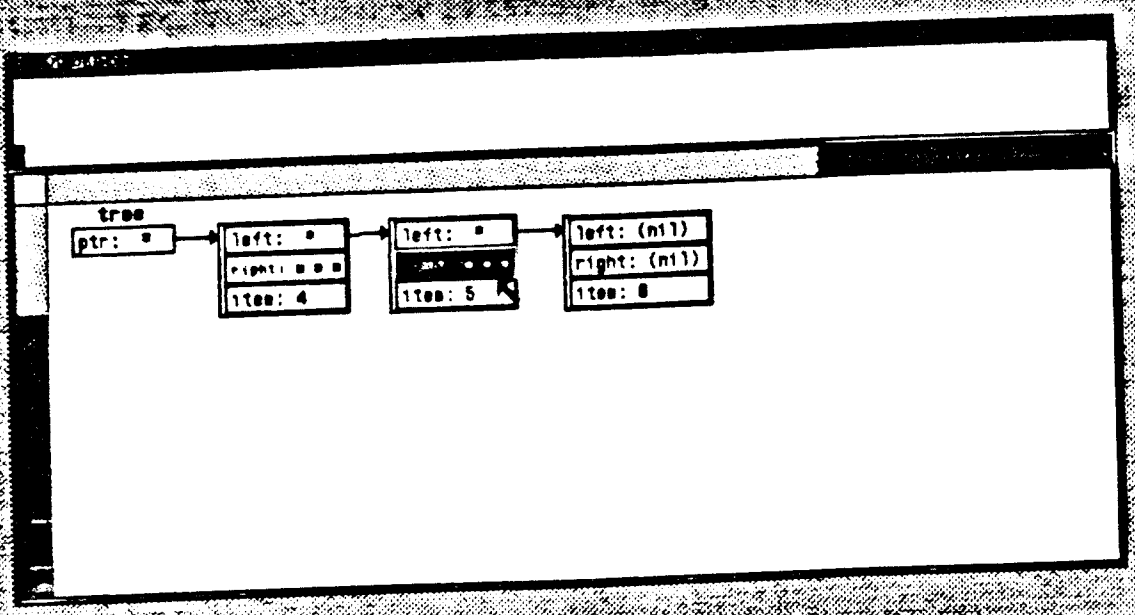
Screen 6.4.2



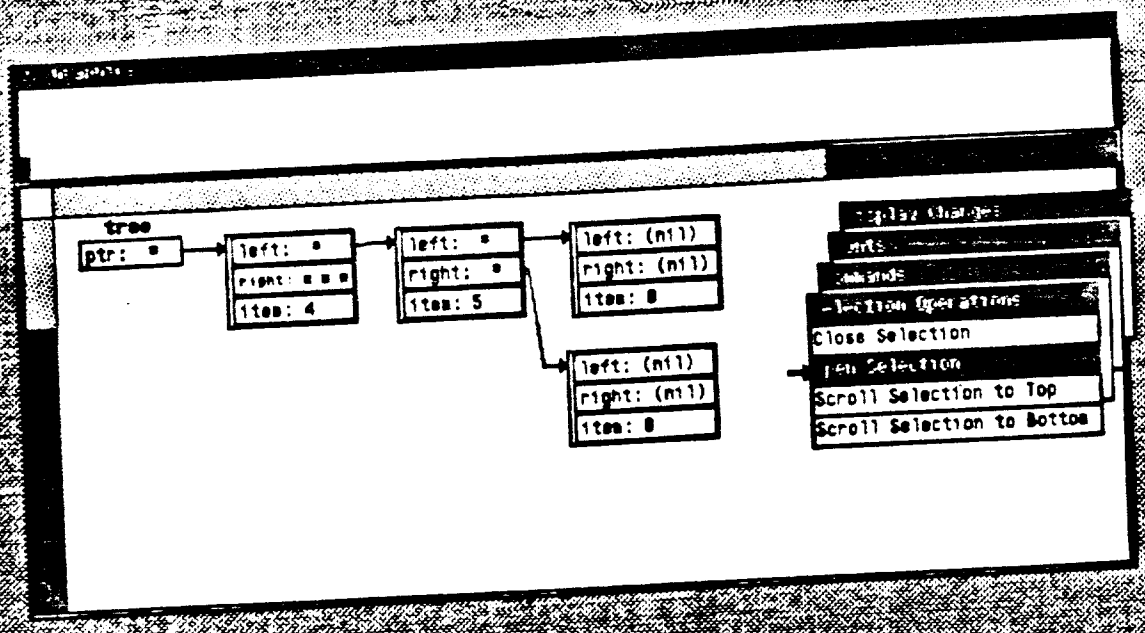
Screen 6.4.3



Screen 6.4.4



Screen 6.4.5



Screen 6.4.6

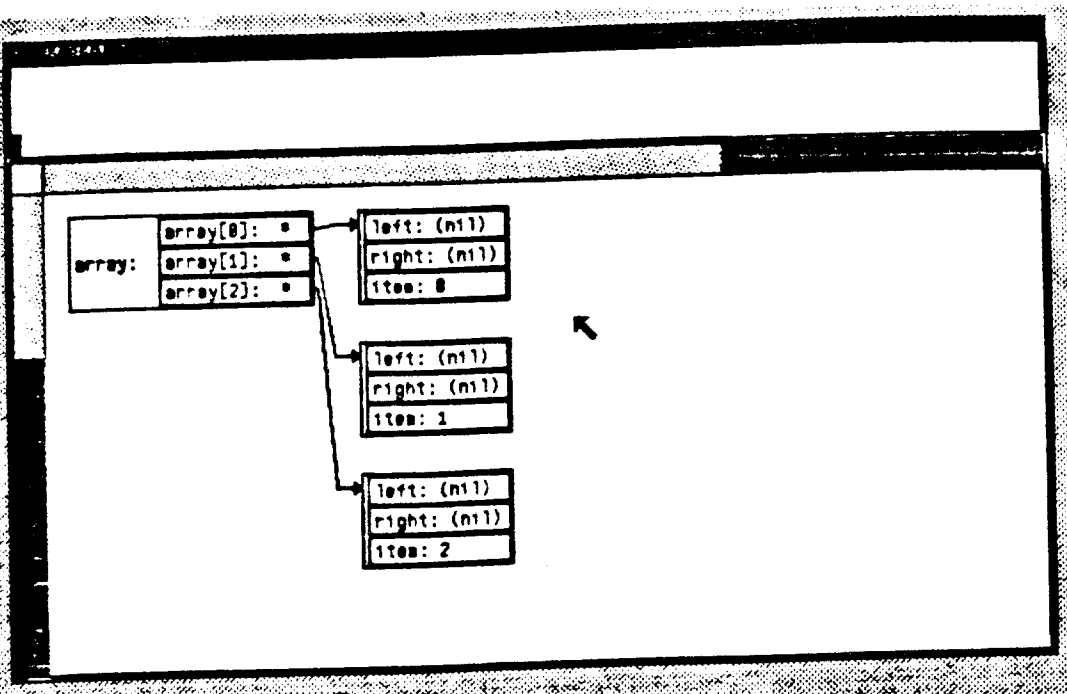
6.5. Structure Movement

The following screens illustrate how the middle mouse button is used to move structures on the screen.

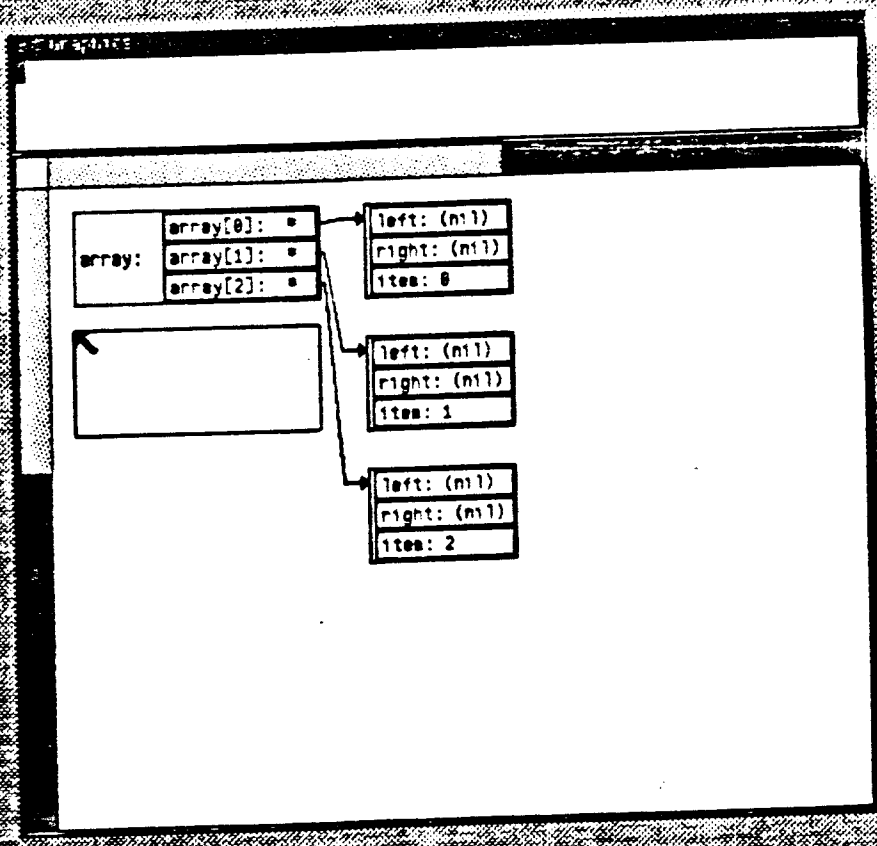
6.5.1 This screen illustrates a layout of an array of pointers created by the positioning algorithm.

6.5.2 To move a structure, the middle button is depressed while pointing to the structure. Here, the array of pointers was pointed to. Moving the mouse while continuing to depress the middle button *drags* an outline of the structure along with the mouse.

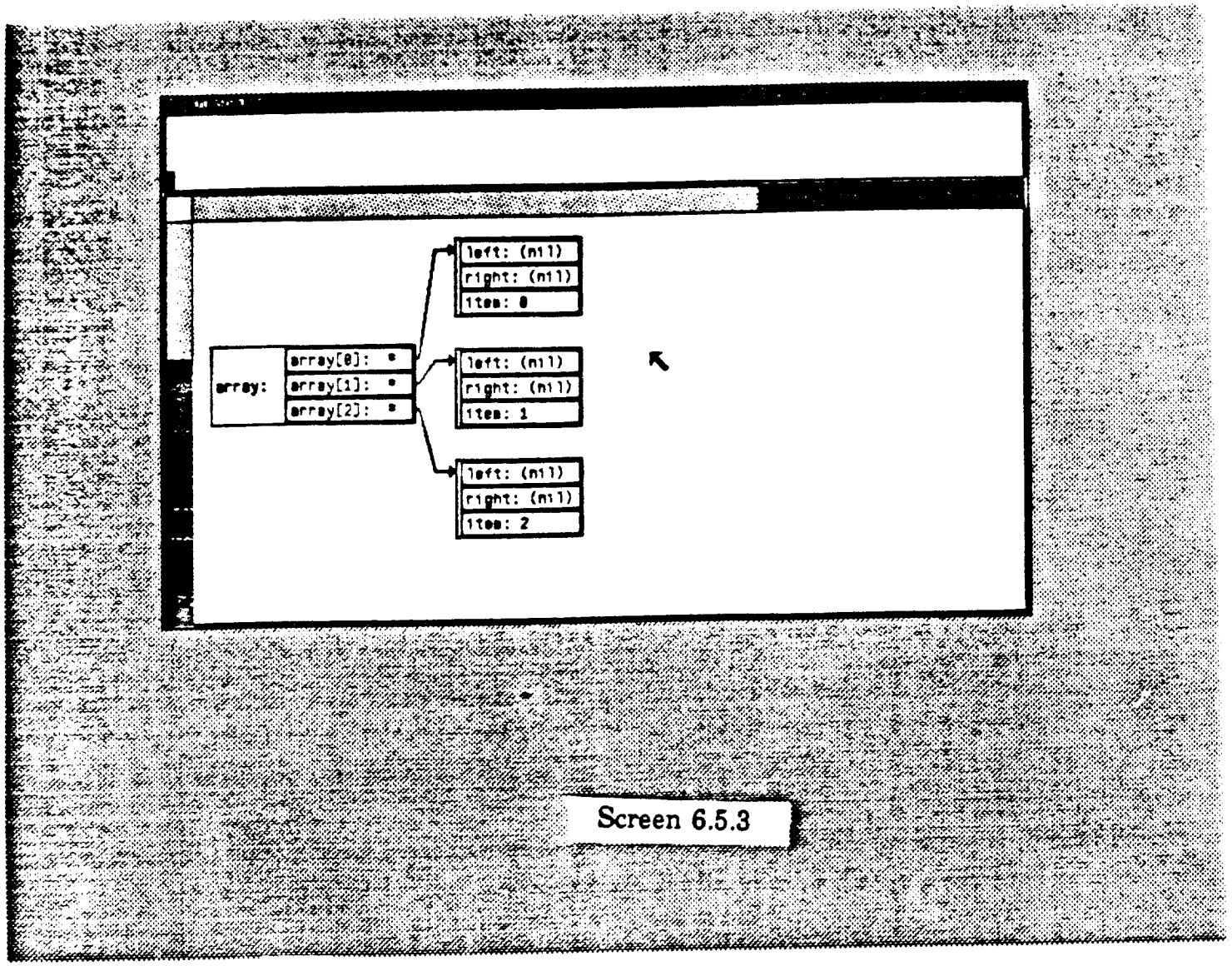
6.5.3 When the middle button is released, the array structure is displayed at its new position.



Screen 6.5.1



Screen 6.5.2


left: (nil)
right: (nil)
item: 0
array:
array[0]:
array[1]:
array[2]:
left: (nil)
right: (nil)
item: 1
left: (nil)
right: (nil)
item: 2

Screen 6.5.3

6.6. Arrow Movement

The following screens illustrate how an arrow, which represents a pointer, can be moved. The actual value of the pointer, as well as the screen representation, is modified by moving the arrow to point to a different object on the screen.

6.6.1 This screen presents three *tree* variables, each of which has two fields that are pointers to other *trees*. The fields are currently *nil*.

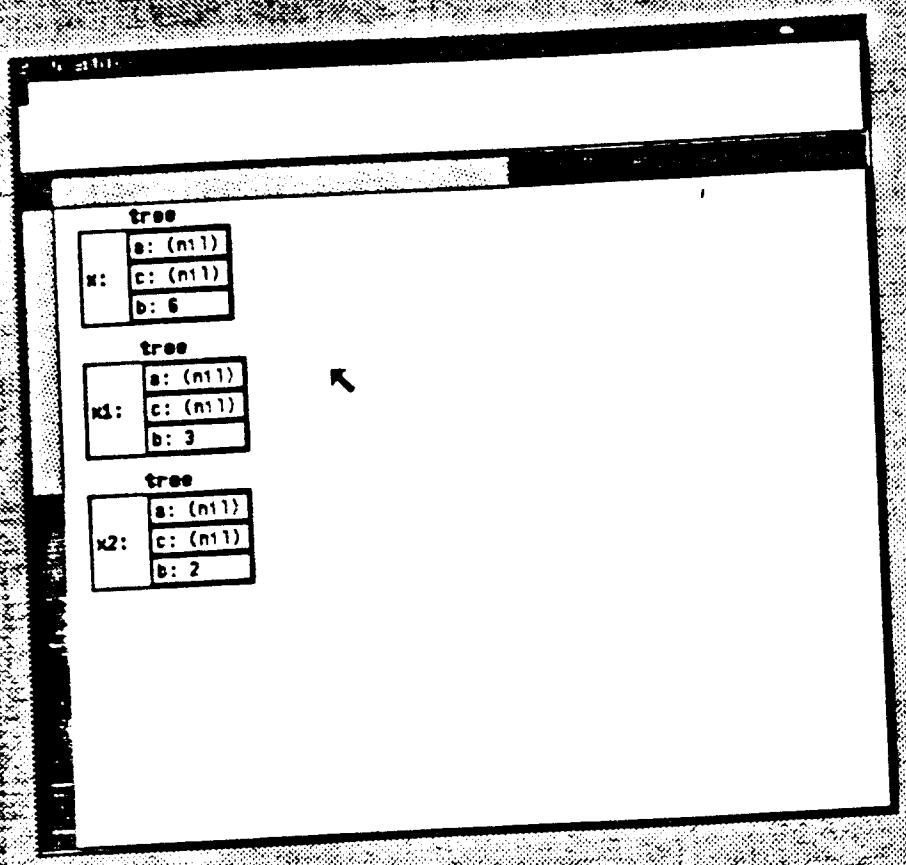
6.6.2 To move an arrow (or to create an arrow if the current value is *nil*), the box holding the pointer is first selected using the left mouse button. Here, the *a* field of variable *x* is selected.

6.6.3 If the middle button is depressed when a pointer field is the current selection, the arrow will be moved. This screen illustrates a new arrow following the motion of the mouse.

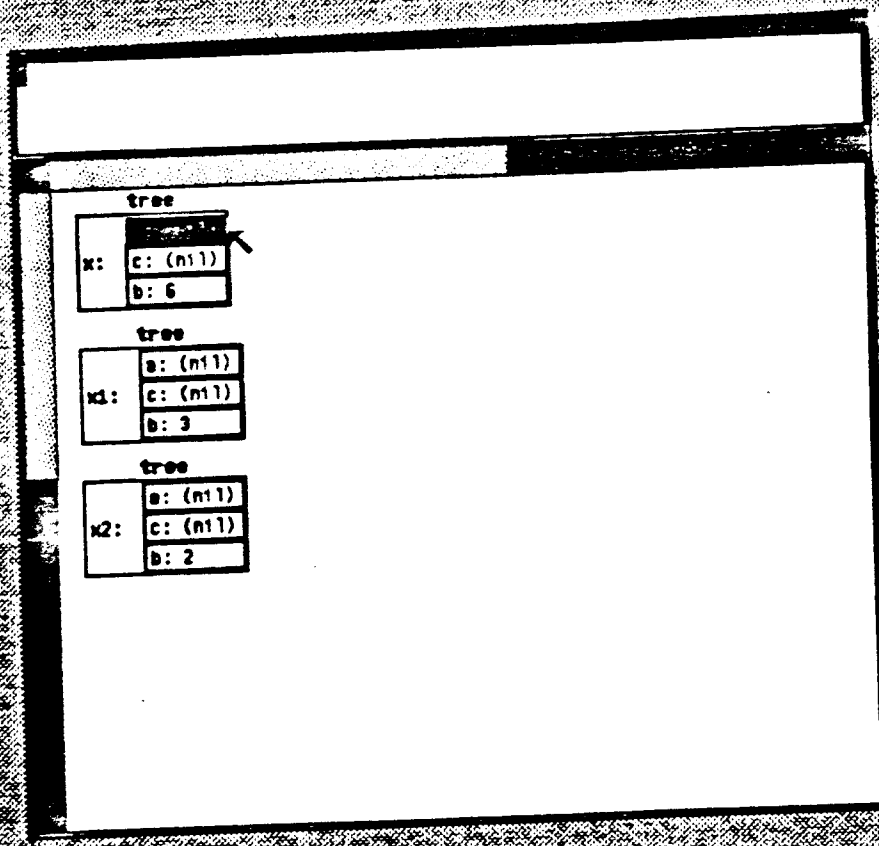
6.6.4 The new arrow is positioned to point to the structure *x1*.

6.6.5 If the arrow is positioned to point to a record structure, the user selects which field within the record he would like the arrow to point to. Here, the entire record *x1* is selected. The value of pointer *a* is changed and the screen is updated to reflect its new value.

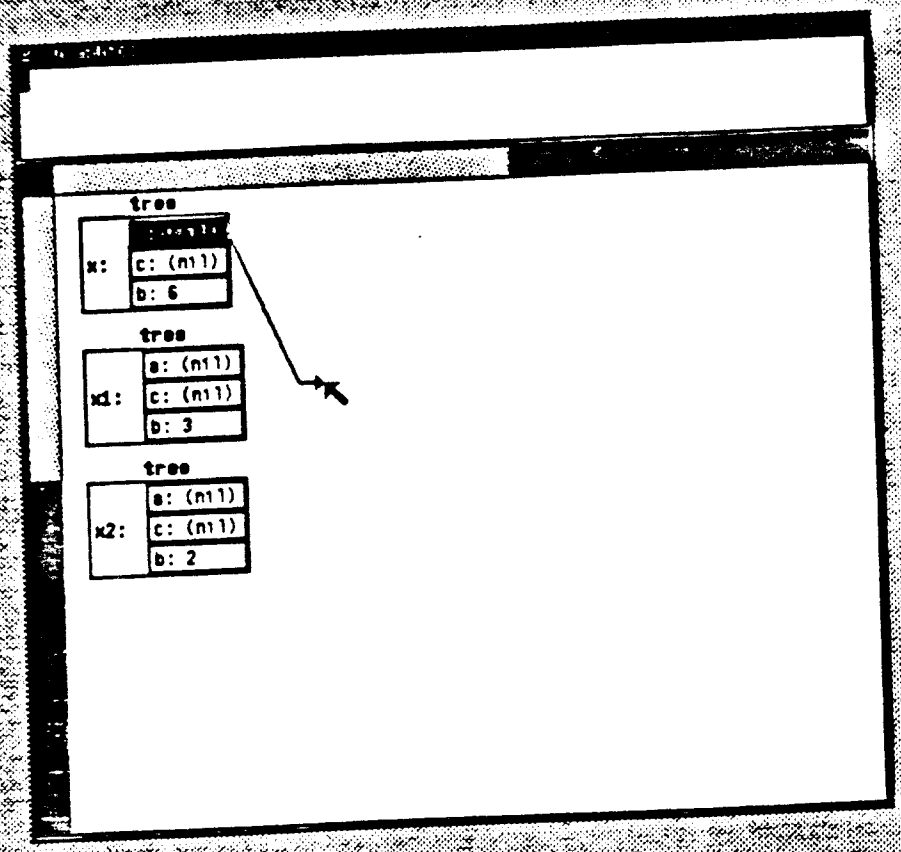
6.6.6 This screen illustrates the three records after field *c* of *x* is made to point to *x2*. Recursive structures may also be defined in this way.



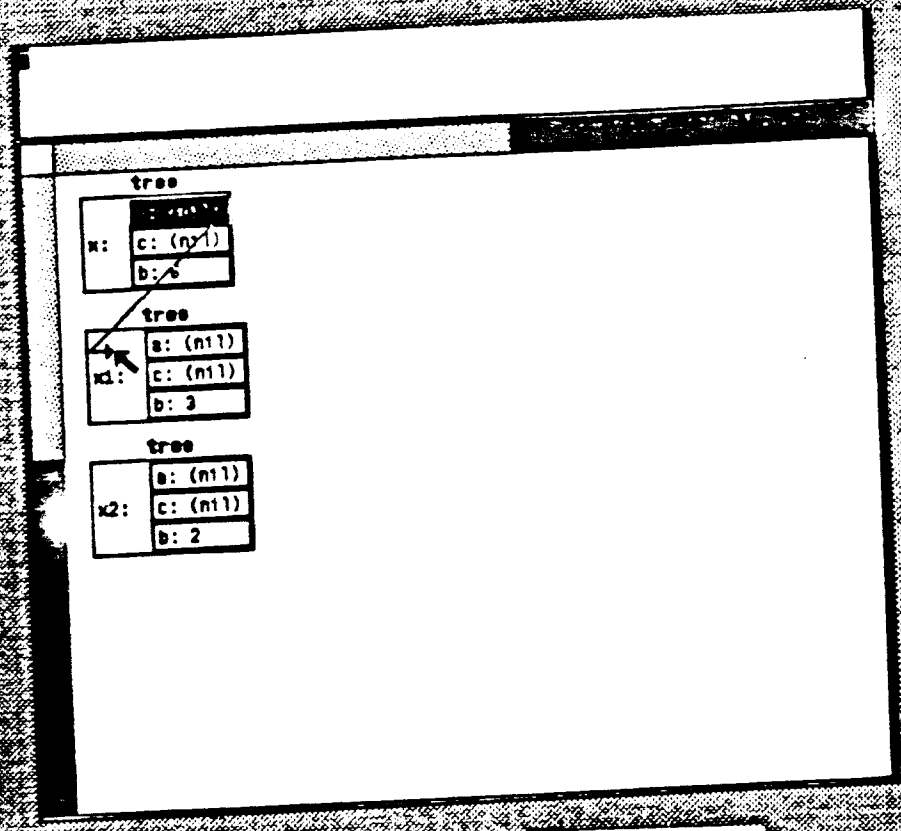
Screen 6.6.1



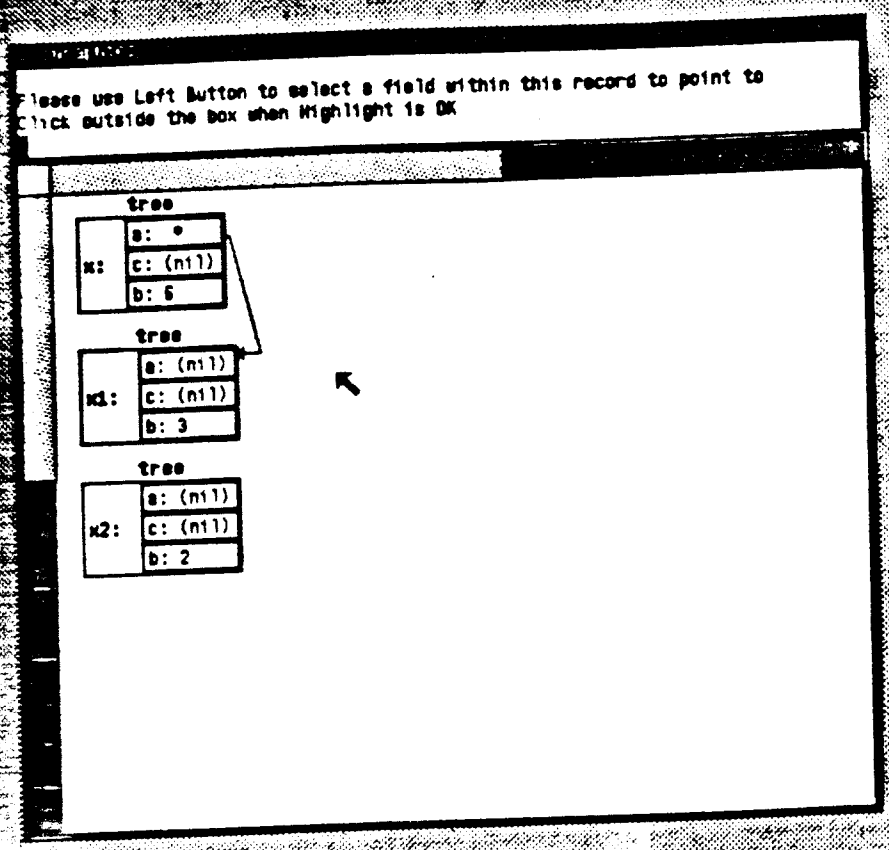
Screen 6.6.2



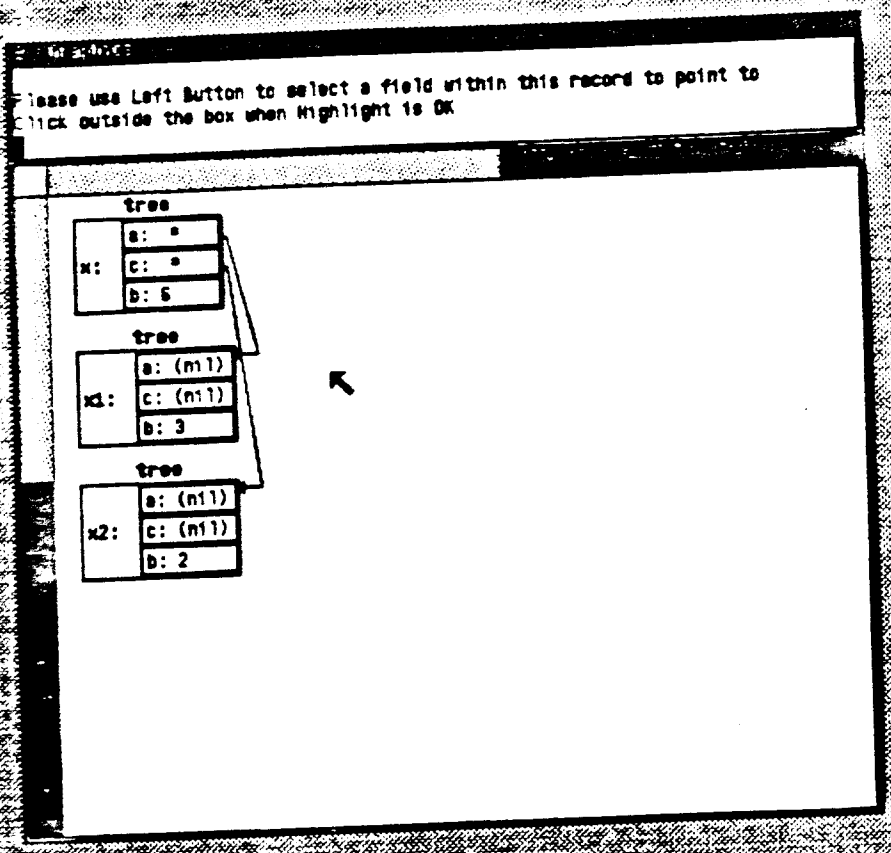
Screen 6.6.3



Screen 6.6.4



Screen 6.6.5



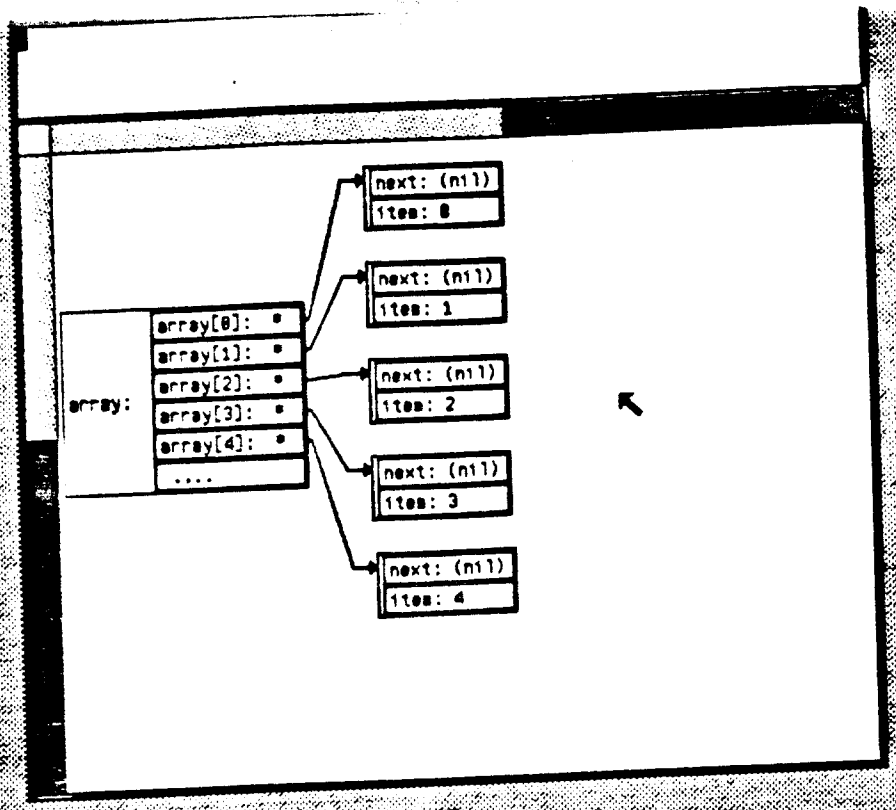
Screen 6.6.6

6.7. Array/Pointer Ellipsis

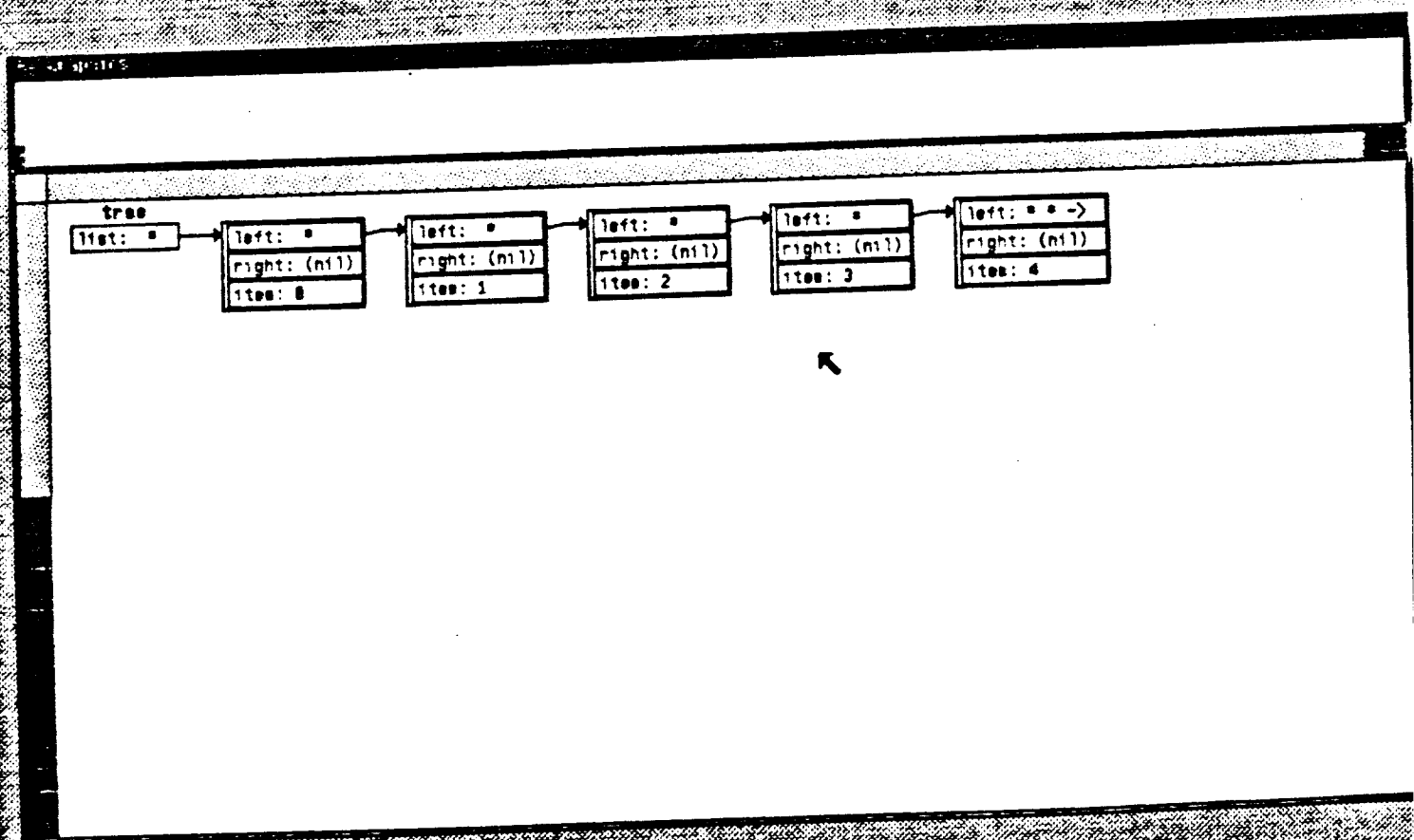
The following screens illustrate ellipsis of array elements and of linked structures beyond a certain depth.

6.7.1 This screen illustrates an array for which only the first five elements are presented. This ellipsis is accomplished by either the ARRAY_SIZE or PRINT_SIZE command. The element value '....' indicates elision. The number of array elements that are *constructed* is set by ARRAY_SIZE. PRINT_SIZE sets the number of array elements to be *presented*.

6.7.2 This screen illustrates a linked list of structures that is displayed only to a depth of five. Ellipsis such as this is accomplished by either the PTR_DEPTH or PRINT_DEPTH command. A pointer value '**→' indicates an elision caused by PTR_DEPTH. A pointer value '***' (not shown here) represents an elision caused by limiting the PRINT_DEPTH of a pointer. The latter elision is equivalent to a CLOSED box. A linked structure thus elided may be OPENED to display subsequent levels.



Screen 6.7.1



Screen 6.7.2

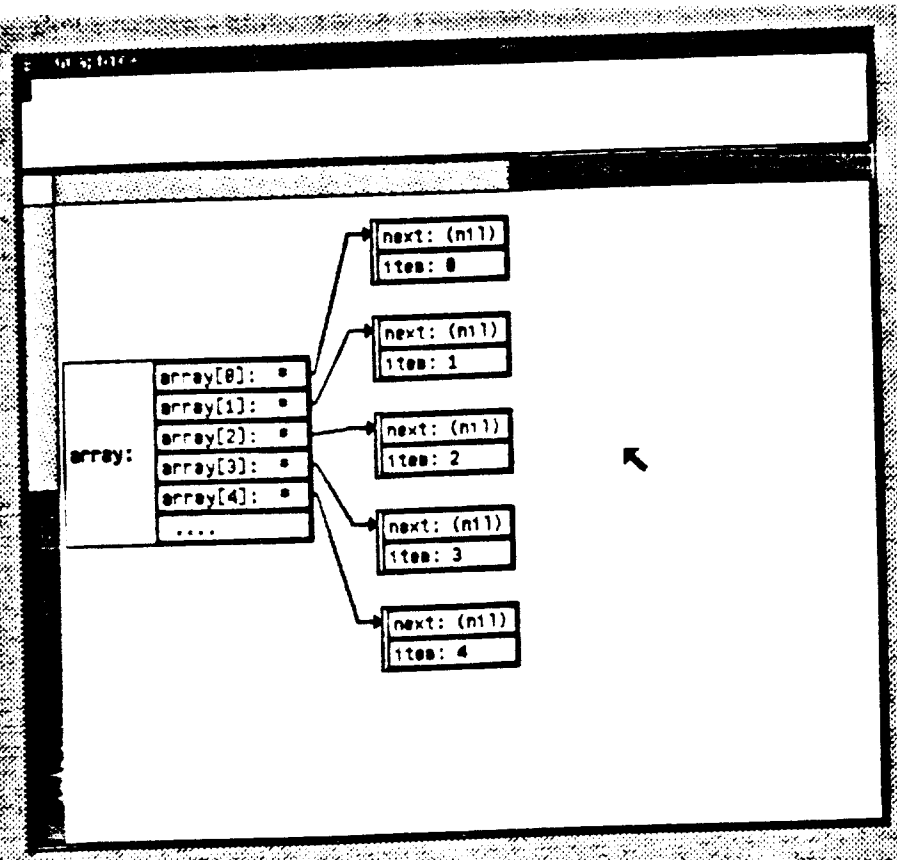
6.8. Array Scrolling

The following screens illustrate how an array may be *scrolled*.

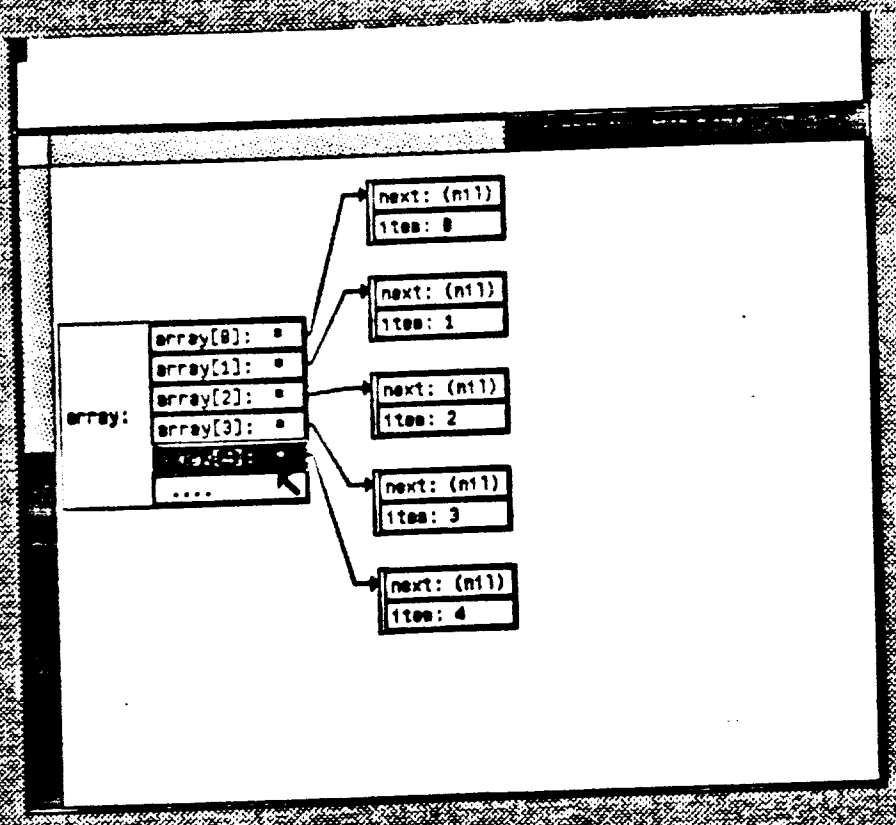
6.8.1 This screen illustrates an array for which only the first five elements are displayed.

6.8.2 Here, an element of the array is selected using the left mouse button.

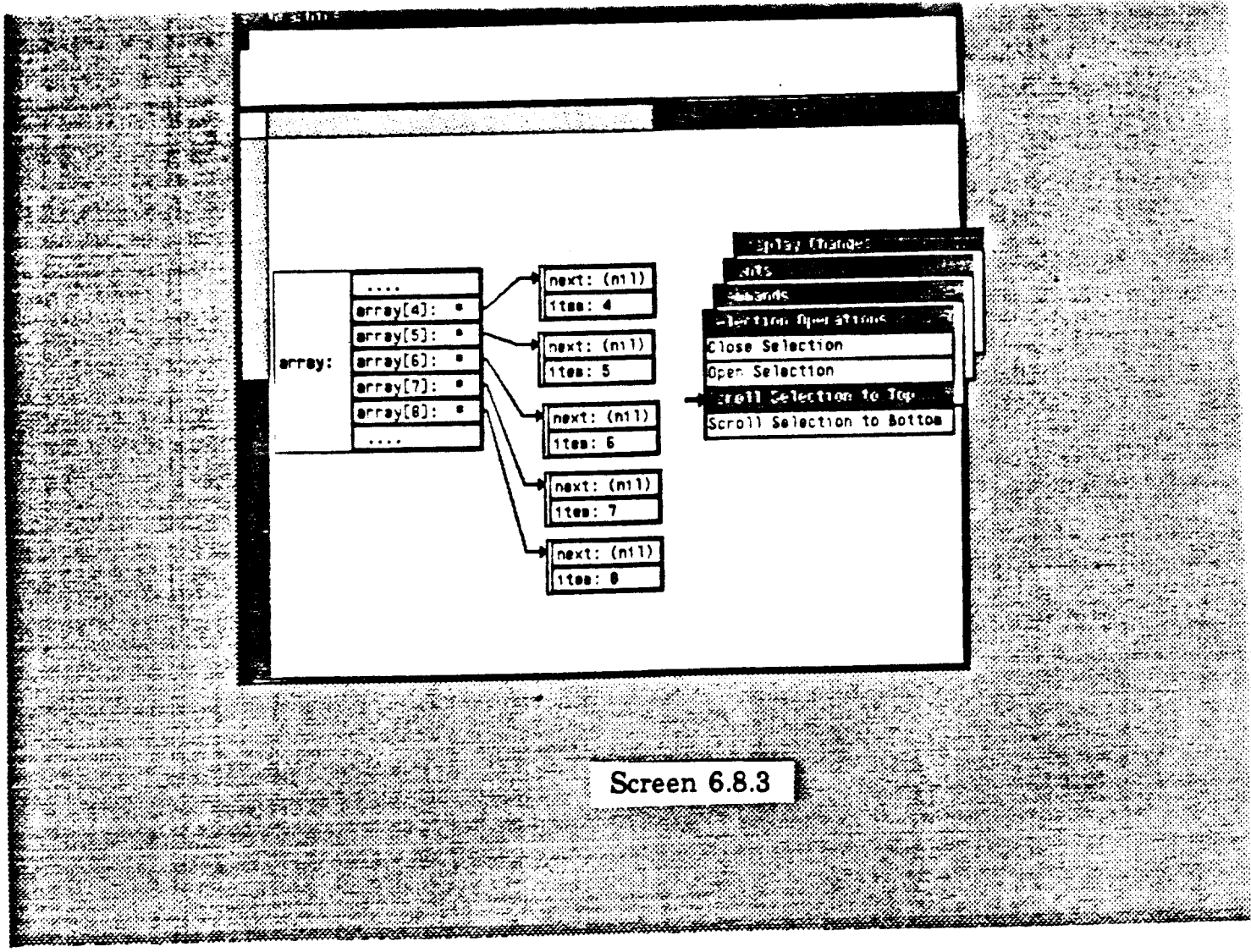
6.8.3 Depressing the right mouse button calls a menu from which the operation SCROLL SELECTION TO TOP is selected. The selected element, number 4, is scrolled to the top portion of the visible array. The beginning element of an array presentation may also be set using the command PRINT_BEGIN. PRINT_SIZE sets the number of array elements that are shown.



Screen 6.8.1



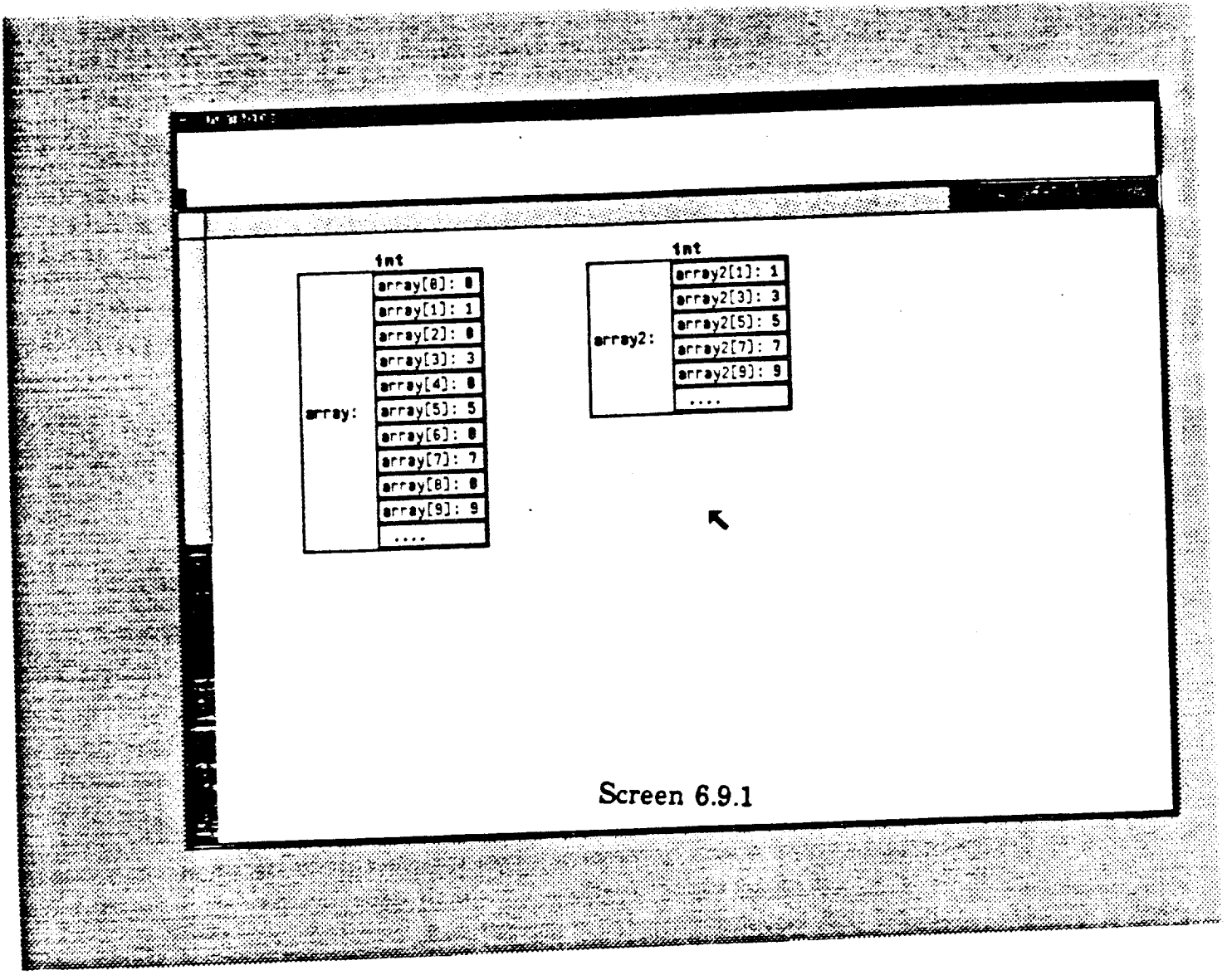
Screen 6.8.2


array:
....
array[4]:
array[5]:
array[6]:
array[7]:
array[8]:
....
next: (nil)
item: 4
next: (nil)
item: 5
next: (nil)
item: 6
next: (nil)
item: 7
next: (nil)
item: 8
Close Selection
Open Selection
Scroll Selection to Top
Scroll Selection to Bottom

Screen 6.8.3

6.9. Array Suppression

This screen pictures two arrays of identical values. The left array is presented normally. All values are show, up to the portion that is elided due to PRINT_SIZE. In the right array, the 0-valued elements of the array are elided by the command SUPPRESS.


int
array:
array[0]: 0
array[1]: 1
array[2]: 0
array[3]: 3
array[4]: 0
array[5]: 5
array[6]: 0
array[7]: 7
array[8]: 0
array[9]: 9
....
int
array2:
array2[1]: 1
array2[3]: 3
array2[5]: 5
array2[7]: 7
array2[9]: 9
....

Screen 6.9.1

6.10. Scroll Bars and Font Changes

The following screens illustrate how large data structures, or many data structures, may be viewed on a small screen. The visible screen can be *scrolled*, presenting different portions of the virtual screen. Alternatively, the font in which data structures are shown may be changed to shrink or enlarge the presentation.

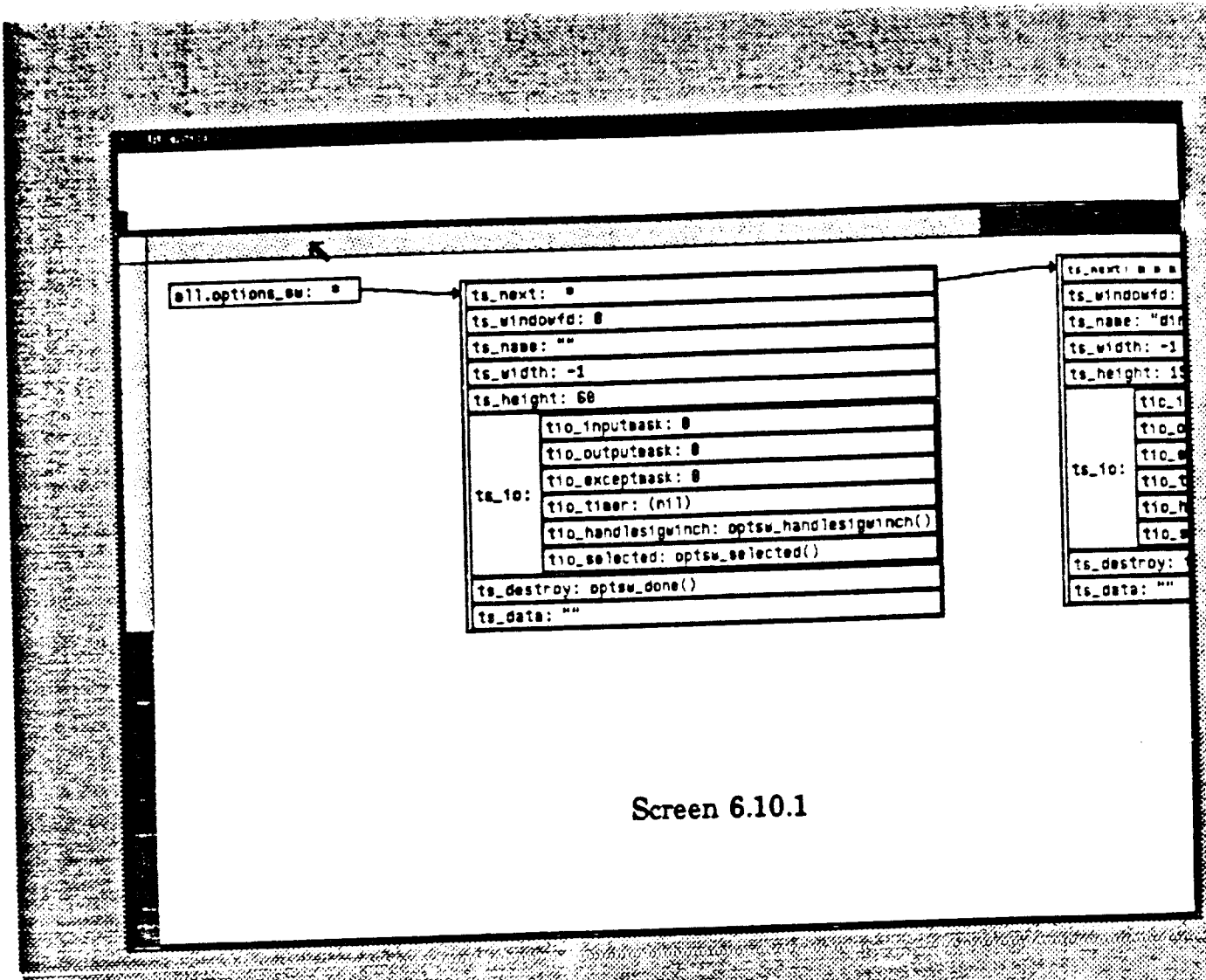
The data structure shown is that of an *option_subwindow*, a standard subwindow structure in *SunWindows*.

6.10.1 This screen shows a linked *option_subwindow* structure that does not fit on the screen. The mouse is positioned in the top horizontal scroll bar.

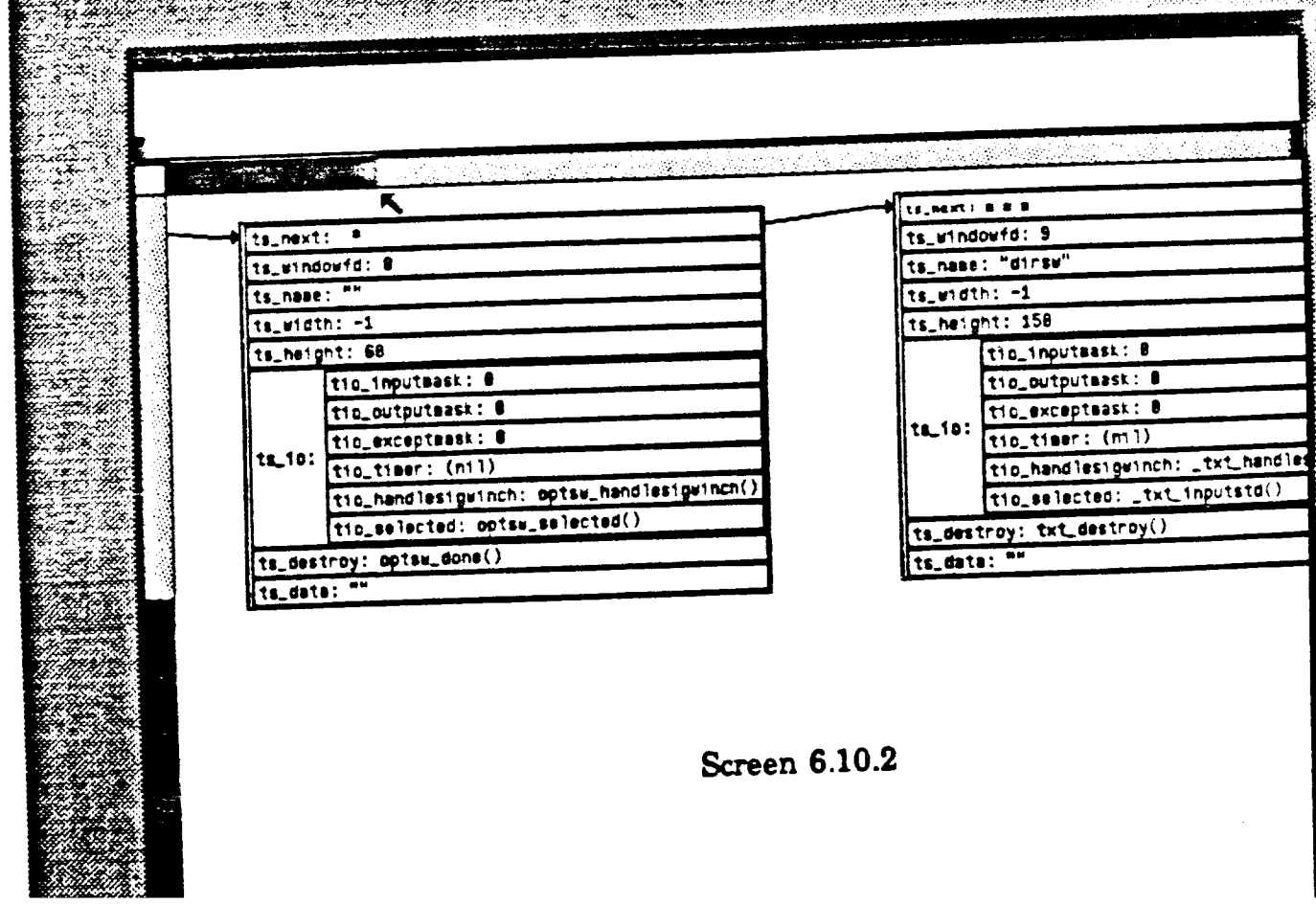
6.10.2 After clicking the left button, the portion of the virtual screen presented is shifted. The presented portion of the screen now begins at the mouse's position in the scroll bar.

6.10.3 Here, a smaller font is chosen. With the entire data structure scaled down, the linked *option_subwindow* fits inside the visible portion of the screen.

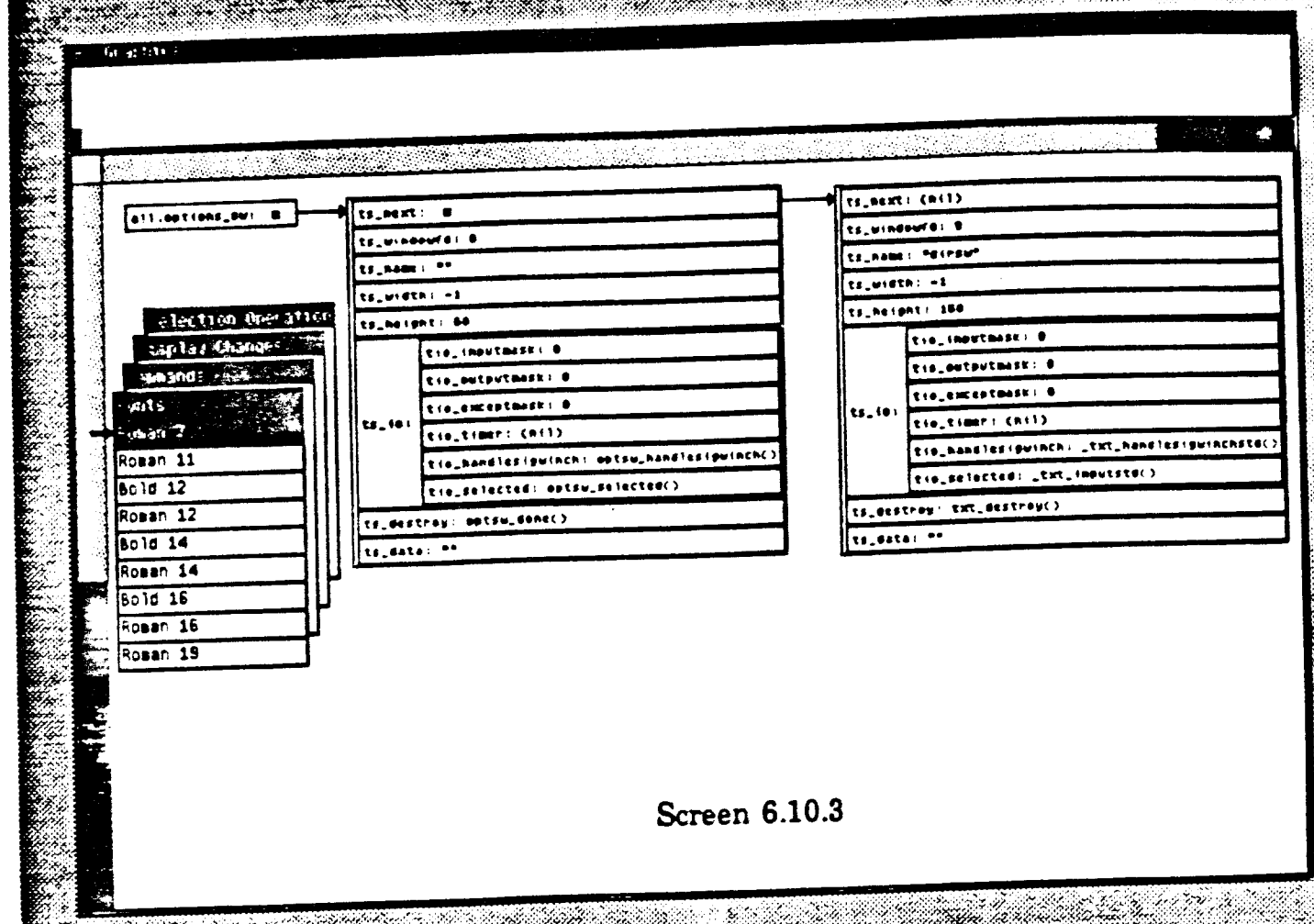
6.10.4 This screen illustrates the data structure presented in a larger font size.



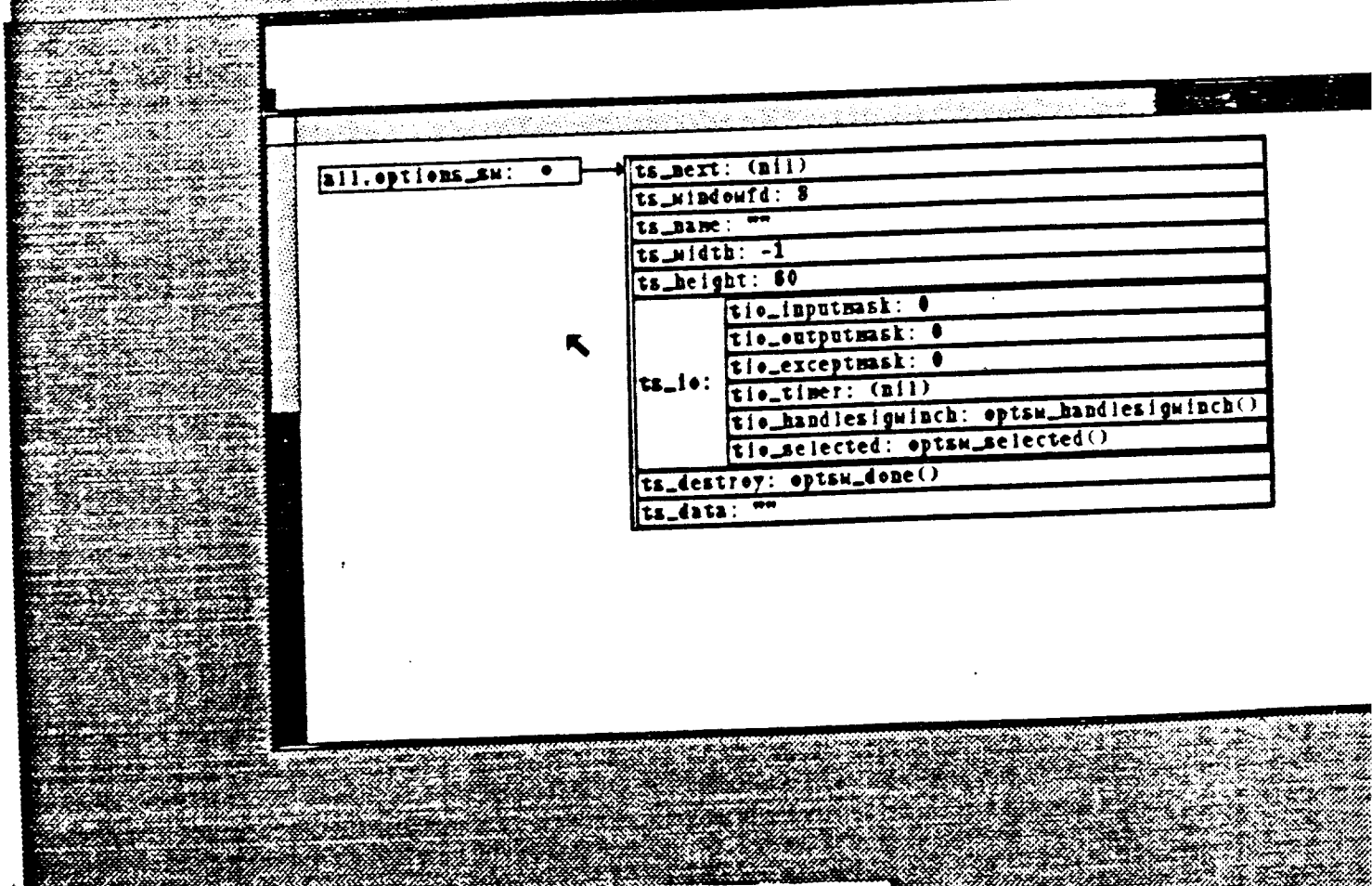
Screen 6.10.1



Screen 6.10.2



Screen 6.10.3



Screen 6.10.4

7. Command Description

Commands to present or modify data structures graphically may come from the keyboard or the mouse. Commands entered at the keyboard will be referred to as GDBX commands. GDBX commands include the standard DBX commands which are unmodified. GDBX commands can be issued before beginning to debug a program interactively. Mouse commands operate on structures already displayed.

7.1. GDBX Commands

GDBX commands may be issued during interactive debugging, or may be placed in an initialization file. If the commands are placed in the startup *.dbxinit* file, they are read and processed at the beginning of each GDBX session. This mechanism can be used to define an environment tailored to an individual user. (The same mechanism is used in the original DBX.) If the commands are placed in a file named *.dbxfile* (where *file* is the name of the program to be debugged) they are read and processed only when this particular program is debugged. This second usage creates an environment tailored to a particular program and the data structures within it.

7.1.1. Presentation Commands

DISPLAY variable_name [, variable_name]

DISPLAY presents the variable's value on the screen. The value is presented as a box structure. This DISPLAYed variable is placed on a list of variables whose values are presented each time execution stops, after a STEP, NEXT, or CONTINUE command. The position of DISPLAYed data structures is maintained across execution steps.

UNDISPLAY variable_name [, variable_name]

UNDISPLAY clears the variable's value from the screen and removes it from the list of variables that are updated after each execution step.

PRINT variable_name [, variable_name]

PRINT presents the variable's value on the screen in the same manner as DISPLAY. PRINTed variables are erased from the screen after each execution step.

ERASE variable_name [, variable_name]

ERASE clears the variable's value from the screen. The entire screen can also be cleared (see CLEAR below).

7.1.2. Layout Commands

ACROSS structure_name [, structure_name]

ACROSS causes pointers to succeeding boxes of a linked structure to point *across* the screen. The default direction is ACROSS.

DOWN structure_name [, structure_name]

DOWN causes pointers to succeeding boxes of a linked structure to point *down* the screen.

7.1.3. Commands Limiting Construction or Presentation

In presenting a data structure on the screen, GDBX first *constructs* boxes to represent the data structure's value and then *presents* these boxes on the screen. The user can control the amount of information GDBX processes at either the construction or presentation phase. Limiting the amount of information *constructed* by GDBX increases the speed with which data structures are presented and conserves memory. Limiting the amount of information *presented* on the screen increases presentation speed and decreases the amount of information shown on the screen.

Construction is done only in response to a PRINT or DISPLAY command. Only *constructed* boxes can be *presented*.

There are three data structure types that can grow very large for which limitation may be important: records, arrays, and linked (pointer) structures. GDBX commands exist to limit processing of these three types during construction and presentation phases.

7.1.3.1. Records

SHOW UNSHOW are used to specify the fields of a record for which information is *constructed*. Without any specification, all fields of a record are constructed. The mouse commands OPEN/CLOSE (see below) control which fields are *presented* on the screen.

SHOW field_name [, field_name]

SHOW is called with a field_name of a record structure type (e.g., SHOW tree.right). SHOW causes a box for the given field to be *constructed* the next time a variable of this type is PRINTed or DISPLAYed. Only fields that are SHOWn will be constructed; all fields not explicitly SHOWn will be suppressed. The SHOW command is used to focus attention on a few particular fields of a large record structure. SHOW's affects all variables of the given type.

UNSHOW field_name [, field_name]

UNSHOW is called with a field_name of a record structure type. A box will not be *constructed* (or *presented*) for this field the next time a variable of this type is PRINTed or DISPLAYed. The UNSHOW command is used to suppress particular fields that are not of interest. SHOW and UNSHOW are not exact antonyms: SHOW entails that fields left unspecified will be *suppressed*, whereas UNSHOW entails that unspecified fields will have boxes *constructed*.

7.1.3.2. Arrays

ARRAY_BEGIN/ARRAY_SIZE control the starting element and number of array elements for which boxes are *constructed*. PRINT_BEGIN/PRINT_SIZE control where the *presentation* of array elements begins and how many elements are shown. An array can also be *scrolled* (see SCROLL SELECTION below) to change the beginning element. SUPPRESS/UNSUPPRESS control the *construction* of 0-valued array elements.

ARRAY_BEGIN array_name integer

ARRAY_BEGIN sets the element number from which *construction* of boxes representing array elements begins. The default starting element number is 0.

ARRAY_SIZE array_name integer

ARRAY_SIZE sets the number of elements for which boxes will be *constructed* for the given array variable. The default array size is 10.

PRINT_BEGIN array_name integer

PRINT_BEGIN sets the element number from which the *presentation* of an array begins. The default beginning array element is 0. The element from which array presentation begins may also be controlled by *scrolling* array (see SCROLL SELECTION below).

PRINT_SIZE array_name integer

PRINT_SIZE sets the number of elements that will be *presented* for the given array variable. The default print size is 10.

SUPPRESS array_name

SUPPRESS instructs GDBX to suppress *construction* of array elements whose values are 0 or *nil*.

UNSUPPRESS array_name

UNSUPPRESS instructs GDBX to *construct* even those array elements whose values are 0 or *nil*. This is the default condition.

7.1.3.3. Linked (Pointer) Structures

PTR_DEPTH controls the number of pointer levels *constructed* whereas PRINT_DEPTH controls the number *presented*. Elision due to limiting PRINT_DEPTH is equivalent to a CLOSEd box.

PTR_DEPTH ptr_name integer

PTR_DEPTH sets the number of pointer levels that will be *constructed* for the given pointer variable. The default pointer construction depth is 5.

PRINT_DEPTH ptr_name integer

PRINT_DEPTH sets the number of pointer levels that will be *presented* for the given pointer variable. The default print depth is 5. The number of levels presented may also be controlled by OPENING or CLOSING pointers.

7.2. Mouse Commands

The following operations are initiated from the 3-button mouse when it is pointing into the graphics window.

7.2.1. Left Button

Clicking the left button while pointing into a structure *selects* that structure or a field within it. The structure selected is called the *current selection*. Multiple clicks in the same structure will cycle through nested fields within the structure. A selected structure can then be acted upon (see the *Selection* Menu below). Clicking the left button outside of a structure clears the *current selection*.

Clicking the left button while it is in the horizontal or vertical scroll bar causes a new portion of the virtual screen to be presented. In the scroll bar, the dark gray rectangle represents the size of the entire virtual screen; the white rectangle represents the currently displayed portion. The new portion to be displayed is determined by the position of the mouse within the scroll bar when it is clicked.

Clicking the mouse in the top left hand box resets the portion of the screen presented to be the upper left-hand corner.

7.2.2. Middle Button

The middle button is used to move structures or arrows. If a pointer value is not the *current selection*, depressing the middle button will move the structure to which the mouse is pointing. Moving the mouse while holding the middle button down *drags* the structure along with the mouse. The structure is re-positioned to the mouse's position when the middle button is released.

If a pointer value is the *current selection*, depressing the middle button moves the arrow emanating from the selected box. The structure pointed to by the mouse when the button is released becomes the new value of the pointer. If a record structure is pointed to, the left button is used to select which field the arrow should point to.

7.2.3. Right Button

Depressing the right button presents several banks of menus. The banks and their commands are:

7.2.3.1. General Commands

• CLEAR: Clears the entire screen of both PRINTed and DISPLAYed structures. Sets the portion of the virtual screen presented to be the upper left-hand corner.

• QUIT: Closes the widow and causes GDBX to exit.

7.2.3.2. Selection

The following commands operate on the *current selection* (see Left Button above).

• OPEN: Causes the selected structure, field, or array element to be presented normally.

• CLOSE: Causes the selection to be presented as a single box. If the box is a simple value it is presented in small font. If the box is a record structure, its substructure is replaced by the string *'record'* displayed in small font. If the box is a pointer, '***' is presented in the box, indicating a suppressed pointer.

• SCROLL SELECTION TO TOP: If the selected box is an array element, the array is *scrolled* so that this element is displayed at the top of the array. This is equivalent to the ARRAY_BEGIN command.

• SCROLL SELECTION TO BOTTOM: If the selected box is an array element, the array is *scrolled* so that this element is displayed at the bottom of the array.

7.2.3.3. Fonts

A font with which to present values within boxes is selected. When a new font is chosen, the graphics window is cleared and all DISPLAYed variables are presented in the new font.

7.2.3.4. Show Changes

This series of commands emphasizes, by blinking, changes to the values of DISPLAYed variables that occurred over the last execution step.

- BLINK NEW: The new values, i.e., the values different from the previous values, of the DISPLAYed data structures are made to blink.

- BLINK OLD: The old values of the DISPLAYed data structures are made to blink.

- BLINK OLD/NEW: The values within the data structures DISPLAYed are made to blink between previous and current values.

8. Implementation

8.1. Overview and Processes

The graphic version of DBX (GDBX) is implemented using three processes. GDBX is the initial program executed. GDBX_TOOL is forked by GDBX, and GDBX_SHOW is in turn forked by GDBX_TOOL. Integration with DBXTOOL requires a multiple-window implementation for GDBX. Multiple windows, in turn, requires multiple processes, because in *SunWindows* each window has an underlying process. The processes communicate using the UNIX pipe mechanism, and synchronize using the SELECT system call. Following is an overview of the three processes.

- GDBX

GDBX is executed from the shell with the command DBX -G or is forked by DBXTOOL (see *Integration with DBXTOOL* below). GDBX creates a socketpair (two pipes), then forks GDBX_TOOL, passing to it the file descriptors of the socketpair. After the socket has been formed, GDBX does a SELECT system call and waits for input either from the keyboard or the mouse.

The GDBX commands PRINT and DISPLAY create a structure that represents the value of the variable (data structure) to be displayed on the screen. After the size of the data structure and its position on the screen have been determined, commands to present the data structure are sent to GDBX_SHOW across the pipe. Other GDBX commands, such as ACROSS/DOWN, SHOW/UNSHOW, PTR_DEPTH, or ARRAY_BEGIN/ARRAY_SIZE, affect the construction of the data structure representation or its positioning on the screen. Original DBX commands, unrelated to presenting a variable's value, are unaffected by GDBX.

Input from the mouse is received by GDBX through the pipe from GDBX_SHOW. This input is interpreted by GDBX and the appropriate response, usually a screen update, is calculated and sent back to GDBX_SHOW for display.

GDBX maintains all the information about the data structures displayed and about the display itself, such as the font size and the part of the virtual screen currently displayed.

• GDBX_TOOL

GDBX_TOOL is forked by GDBX. This process opens a window on the screen, and creates a tty and a graphics subwindow. GDBX_TOOL passes the socketpair file descriptors it has received from GDBX on to GDBX_SHOW when it forks this new process. After creating GDBX_SHOW, GDBX_TOOL is no longer involved in GDBX processing. However, it does retain global tool functions such as CLOSE, which turns the window into iconic form, and QUIT, which ends the process.

• GDBX_SHOW

GDBX_SHOW is forked by GDBX_TOOL. It receives a pair of file descriptors through which it communicates with GDBX. GDBX_SHOW takes over the graphics subwindow created by GDBX_TOOL. GDBX_SHOW is responsible for displaying data structures in this subwindow and for sending mouse input from this window to GDBX.

To present data structures on the screen, GDBX_SHOW first initializes a set of *pixrects* (pixel rectangles, bit patterns held in internal memory). These pixrects are written according to commands received from GDBX. When a data structure has been completely written to a pixrect, the contents of the pixrect are written to the visible screen at a single instant.

Mouse input from the graphics window consists of a button type (left, middle, or right) and a set of x,y coordinates or menu selection. Mouse input is sent to GDBX, which processes it and sends back commands to appropriately update the screen.

8.2. Integration with DBXTOOL

GDBX was designed to be integrated into the DBXTOOL product of Sun Microsystems. DBXTOOL is a multiple subwindow tool which runs DBX inside one subwindow. The subwindows present the following information: a status subwindow names the source file displayed; a text subwindow presents the source file context of the present stopping point; a panel shows DBX commands; a tty subwindow runs DBX; and a text subwindow displays values of variables.

The graphics subwindow of GDBX_TOOL and GDBX_SHOW replace the last text subwindow.

The following aspects of GDBX were influenced by the decision to integrate GDBX into DBXTOOL.

- It was decided to make GDBX_SHOW operate in a separate graphics window rather than attempt to use the fifth subwindow of DBXTOOL. The sizes of DBXTOOL's four other subwindows severely constrain the size attainable by the fifth subwindow. This constraint is due to subwindows inside a *SunWindows* tool being tiled, rather than overlapped, and because each subwindow in DBXTOOL has an absolute and practical minimum size. Rather than restrict users to a small tiled window area, an overlapping window was implemented by creating an independent tool (GDBX_TOOL) and having GDBX_SHOW take over the graphics subwindow within this tool.

- Since GDBX will be forked by DBXTOOL and will operate within the tty emulator of the fourth subwindow, GDBX was designed to remain a normal shell command. An alternative would have been to fork GDBX from GDBX_TOOL, whereupon GDBX itself could take over the graphics subwindow of GDBX_TOOL. However, this would have meant that GDBX could only be run from inside a graphics tool, making it less general and making it impossible to integrate with DBXTOOL. The process hierarchy described in this paper was necessitated by the desire to integrate GDBX into DBXTOOL.

The following sections describe in more detail, by process and chronology, the algorithms that present data structures graphically.

8.3. Box Construction Algorithm

When a command to display a variable's value (a data structure) is given, GDBX constructs a representation of the value that is a linked *box* structure. All further graphics processing takes place upon this box structure. Each box is an object that will later be physically presented on the screen to portray part of the data structure. Each box holds a value, memory address, name, size in pixels, and an x,y position on the screen. Associated with the box are user-definable characteristics that affect the *construction* of the box representation or its subsequent *presentation* on the screen. The commands SHOW, PTR_DEPTH, ARRAY_BEGIN/ARRAY_SIZE, and SUPPRESS/UNSUPPRESS set a box structure's *construction* characteristics, while the commands OPEN/CLOSE and ACROSS/DOWN affect a box's *presentation*.

The box structure is linked together as follows: A nested data structure is represented by a box with links to *child* boxes. A box that has no parent and that appears as the outermost box when displayed will be referred to as a *top-level* box. A record structure (in C or Pascal) is composed of a top-level box with a pointer to a linked list of child boxes each representing a field of the record. An array is also represented in this way, with each child representing an array element.

A box representing a pointer has a link to the box representing the data structure it points to. Boxes that are pointed to have links to each box which points to them.

The algorithm that creates the box structure is a revision of the DBX print routine. Where the original DBX prints a value to the terminal, the GDBX routine creates a box into which the value is printed (as a string). The memory address of this value is calculated and associated with the box. The box is then linked to those boxes that define its context.

There are several interesting additional modifications to the DBX print algorithm that were necessary in GDBX:

- In the original DBX, pointers are printed as addresses. In GDBX, boxes representing the structure at this address are recursively constructed. The values beginning at this point in the program's address space must be made available to GDBX. These values (data structures) are pushed onto the GDBX stack, evaluated, and then boxes for them are constructed recursively. These boxes are then linked into the existing box structure.

- Since structures pointed to are evaluated in GDBX, rather than just their addresses being printed, the possibility of infinite recursion is introduced. A hash table of boxes, indexed by memory address, is maintained to prevent such recursion. This box table also ensures that a box that is pointed to by several other boxes will itself appear on the screen only once, with many arrows pointing to it.

  A subtle exception to the above solution occurs with record-type data structures. In record structures, the first *field* of the record has the same location in memory as the top-level *record*. A special provision is made to recognize this case as being non-recursive. In effect, there are two different boxes that share the same memory address.

- Although boxes could hold only the values to be presented on the screen, GDBX adds the *type* of the variable represented to the box as well. This additional information allows operations such as SHOW/UNSHOW, DOWN/ACROSS, to be applied generically to types. In addition, the type of a variable (*int*, *char*, a typedef of a structure, etc.) can be displayed with the top-level box on the screen. Type information could also be used to influence how a data structure is displayed.

8.4. User Controlled Modifications to Box Construction

There are several commands available to the user that modify the *construction* of the box structure described above.

8.4.1. SHOW/UNSHOW

These commands allow the user to specify that a certain field of a record structure should be included in/excluded from box construction. When the boxes representing the fields of a record are being constructed, the *show* field of the variable's type is consulted. If the value is *unshow*, the corresponding box is not constructed and not linked to the other fields' boxes.

8.4.2. ARRAY_BEGIN/ARRAY_SIZE

The user may set the element number from which array construction begins, or may limit the number of elements for which boxes are constructed. To implement this, the beginning element and size limit associated with the array variable are consulted within the loop that constructs the boxes representing the array elements.

8.4.3. SUPPRESS/UNSUPPRESS

If the user requests that 0-valued elements of an array be suppressed, boxes representing elements whose value is *0* or *(nil)* are unlinked from the chain of boxes representing the array.

8.4.4. PTR_DEPTH

The user may set the *depth* to which he would like pointers for a given type to be constructed. *Depth* is a count of the number of pointer levels emanating from the top structure. To implement the depth limit, a count of recursions due to following a pointer (as opposed to a nesting) is maintained. Recursion is terminated when this count is above the

depth limit set for this variable type.

8.5. Positioning Algorithm

After the linked boxes representing a data structure have been created, a search is made for a place to present each top-level box. First, the size of the top-level box is determined. The size of each internal box is calculated recursively and summed to give the size of the top-level box. A position on the virtual screen for this top-level box is then found that also defines the positions of all the internal boxes.

The procedure that finds a position for a box is an efficient one that formats linked data structures dynamically and produces reasonable layouts. It does not attempt to calculate specialized or refined layouts. The procedure is given the size of a box, and it returns an x,y position. This routine is not deeply embedded in the GDBX code and may therefore be replaced by a program that specializes in more accomplished layouts.

The user may command a data structure of a given type to be presented either ACROSS or DOWN the screen. Presenting a data structure ACROSS the screen means that pointers to structures will point from left to right and large data structure will grow horizontally. Data structures presented DOWN will grow vertically.

The ACROSS DOWN commands are one example of how a user might define the way in which he would like to format particular data structures on the screen. A more intelligent layout algorithm could incorporate other layout variations, such as a binary tree format, a linked list format, etc. When the layout options are thus extended, the GDBX command interface should also be generalized.

A two-dimensional array is maintained of the parts of the virtual screen that are occupied. A search is made for the first space into which the top-level box will fit. The search for an open space on the screen proceeds as defined by the direction of growth (ACROSS DOWN) of the data structure's type. The size of the rows and columns is determined dynamically by the size of the data structures already on the screen. The search algorithm first attempts to find an open place on the visible portion of the virtual screen.

The position of DISPLAYed variables is maintained across execution steps. DISPLAYing these structures at the same position allows apparently instantaneous update of the data structure's values, and enables changed values to be emphasized (see BLINKing below). The new position of a DISPLAYed data structure that the user has MOVED is also maintained.

The position of PRINTed data structures is not maintained across execution steps. Any number of variables may be PRINTed in any order. Thus, to present PRINTed variables in an orderly fashion, data structures that were PRINTed are cleared from the screen after each execution step. The position of a PRINTed variable is re-calculated for each PRINT request.

8.6. Box Display

A box is displayed by sending GDBX_SHOW the coordinates of the four corners that define

the box, the box's name, and the value to be printed inside. A box is erased by displaying the box again with an XOR raster operation.

Top-level boxes are placed on a *display-list* which is searched when a *selection* is made. A box is displayed with one of three presentation types: PRINT, DISPLAY, or ERASE. The presentation type of the box is consulted to determine whether a box has already been presented or erased, thus terminating recursion. ERASEd boxes are removed from the *display-list* immediately and PRINTed boxes are removed after an execution step.

8.7. Screen Presentation

GDBX_SHOW receives messages from GDBX that indicate the coordinates and values of the box to present. The primitive graphics operations of printing vectors and strings are made to a pixel rectangle (*pixrect*) in memory.

There are four pixrects that GDBX_SHOW manages. Two pixrects are for DISPLAYed boxes. The pixrect into which DISPLAYed boxes are written changes with each execution step, and the previous pixrect's values are preserved. Changes occurring in a DISPLAYed data structure are then easily made apparent by comparing these two pixrect representations through raster operations (see BLINKing below).

Another pixrect is used for PRINTed data structures, which are not re-displayed at each step. This pixrect is cleared on each execution step.

Just before presenting the pixrects to the screen, the current DISPLAY pixrect and PRINT pixrect are written to a *scratch* pixrect which is then written to the screen. This use of a scratch pixrect avoids the undesirable flicker which occurs if two separate pixrects are written to the screen in succession, and makes the screen appear to be instantaneously updated.

The size of the pixrects maintained by GDBX_SHOW is larger than the area of the graphics subwindow on the physical screen. Only a portion of the entire pixrect is actually written to the screen, thus affording a 'virtual window' capability. The portion of the pixrect actually displayed on the screen is changed by clicking the mouse in the vertical or horizontal scroll bar.

8.8. Mouse Operations

8.8.1. Selection

A top-level box, a field within a record structure, or an array element, is *selected* by pointing to the field or element and clicking it with the left mouse button. The x,y coordinates of the mouse position are sent to GDBX. GDBX searches through the boxes on the *display-list* and determines, by comparing coordinates, which field or element was pointed to.

Once the selected box is determined, it is highlighted on the screen by XORing a box of background color over the selected box. The highlighting is removed before the selected box is ERASED or when the left button is clicked outside of any box.

8.8.2. Structure Movement

GDBX allows a user to move data structures to improve the screen display, or to present a data structure as he conceives of it. Structure movement is implemented by first identifying the top-level box that the mouse is pointing to. This identification is done as in *selection*. The box's dimensions are then sent to GDBX_SHOW which continuously presents an outline of the box as it is dragged along, following the mouse. When the middle button is released, the top-level box (and all its internal boxes) is erased, along with the arrows into it and out from it. The new coordinates for the top-level box, are placed into the box and the box is re-presented.

Arrows *out of* the moved box are correctly re-displayed automatically, connecting to boxes already present on the screen. Pointers *into* the moved box are updated by re-displaying each of the boxes that contain pointers to the moved box.

8.8.3. Arrow Movement

Creating an arrow or moving an arrow to point to a different box changes not only the screen representation of the data structures but also the actual value (an address) of the pointer variable.

The implementation of arrow movement is similar to that of structure movement described above. When the middle button is depressed the type of the *current selection* is checked. If the selection is a pointer variable, this fact is sent to GDBX_SHOW, along with the coordinates from which an arrow should begin. While the middle button remains depressed, GDBX_SHOW continually presents an arrow from the *current selection* to the position of the mouse.

When the middle button is released, the mouse coordinates are sent to GDBX which identifies the box to which the mouse is pointing. If the box is a record structure, the user is asked to select a field within this structure, an operation for which the standard selection mechanism is used. The memory location of the box to which the arrow now points is written to the pointer variable in the address space of the program being debugged. Lastly, the representation of the pointer on the screen is updated.

8.8.4. Fonts

A variety of sizes of Roman and Bold fonts are available. Requests for a font change are sent to GDBX, which compensates for changes in height and width when calculating the pixel dimensions of a box. The font in which text is presented is sent to GDBX_SHOW by GDBX as one of the parameters of the text print command.

8.8.5. Blinking

A DISPLAYed data structure is presented after the completion of each execution step. Within a complex data structure, it may be difficult to spot changes that occurred as the result of this execution step (which may consist of many instructions). By using two pixrects, one for the previously DISPLAYed values and one for the currently DISPLAYed values, changes in value can be made to blink, thus rendering them readily apparent.

An AND raster operation of the two pixrects together gives those values that were unchanged by the execution step. Repeated XOR raster operations between the current values and the result of the AND operation makes the new values blink. Through various raster operations, different types of blinking can be obtained.

8.9. Multiple Language Support

GDBX was originally written and debugged using only the C-language facilities of DBX. The language-dependent routines in DBX are isolated in a single file for each language supported. Addition of graphic debugging for Pascal programs was readily accomplished.

9. Performance

GDBX does a considerable amount of work in addition to the fixed overhead of DBX: box data structures are built, information is sent over a pipe, and displays are created on the screen. These additions are substantial and there was concern that performance could be poor.

In fact, the implementation of GDBX displays data structures with an acceptable, even surprising, rapidity. Simple values and small structures are shown almost instantaneously. The presentation of larger records or linked structures have a noticeable lag time of perhaps a second or two. Yet this is little more time than is required to present a structure on a normal tty screen, and far less time than it would take for the user to print out the structures pointed to manually.

The elapsed time and the CPU time consumed by various display operations was measured using the UNIX system call *time*. A script of commands placed in a *.dbxinit* file was used to take the measurements. GDBX and DBX were each run on the same script of commands four times. For scalar values GDBX ran approximately 30% slower than DBX. GDBX required from 2-10 times as long to present structured data.

Submitting commands automatically from a script produced results for DBX and GDBX that could be compared easily. However, this technique does not accurately reflect actual command submission to a debugger when it is being used interactively. When a user takes time to digest the information displayed and to consider his next command, the additional time that GDBX requires to calculate and display a data structure will be far less evident than it is in these performance measurements.

By far the largest additional overhead incurred by GDBX is that of writing display information over the pipe to GDBX_SHOW. For the display of large structures, *writes* consume approximately 50% of the CPU time. Intelligent encoding of display information could reduce this appreciably.

A description and analysis of the types of display operations measured are given below:

1. **Startup/Run**: This script began the debugger, ran the program without breakpoints, and quit without doing any printing or graphic processing. Compared to DBX, the additional time required to start up GDBX, due to forking GDBX_TOOL, GDBX_SHOW, and initializing the window on the screen, was calculated. This additional time was factored out of the following

measurements, whose intention is to determine the time required for displaying objects once processing has begun.

2. **Integers, Characters, Strings** : These scripts DISPLAYed the value of variables of one of these scalar types. The results show that there is an insignificant additional delay incurred by GDBX. CPU time is increased by only 10-15% and elapsed time by approximately 30%. These results imply that the additional overhead of box construction, the placement algorithm, and pipe communication is acceptably low for these fundamental types.

The presentation of these simple types is the only type of display operations that can be directly compared between GDBX and DBX. In the remaining operations, GDBX actually displays more information about the data structures, or displays it in a more meaningful way, than does DBX. Thus, any large performance degradation is mitigated by the value added by GDBX.

3. **Linked Structures**: This script DISPLAYed a variable that is a pointer to a record structure. The number of structures DISPLAYed by GDBX grew from one, when the pointer value was *nil*, to a maximum of five record structures of three fields apiece. The corresponding presentation of the variable's value by DBX always consisted of a single address. The time required for the display of the linked structures by GDBX was approximately twice that required by DBX.

4. **Record Structures**: This script DISPLAYed several complex, nested record structures. In this case, the same amount of information was displayed by GDBX and DBX, but in quite a different format. The time required by GDBX was about triple that by DBX. The SHOW/UNSHOW commands may be used to trim the amount of information displayed to achieve better performance.

5. **Arrays**: The script *Integer Array* DISPLAYed a 10-element integer array, and the script *Pointer Array* DISPLAYed a 10-element array of pointers to record structures. In the GDBX implementation, arrays are cast into a box structure similar to records. This imposes overhead on an array that DBX does not, but allows operations such as OPEN/CLOSE to operate on arrays as well as records.

In *Pointer Array*, GDBX DISPLAYed the structures that are pointed to by each of the array elements, whereas DBX printed only the structure's address. This display of additional information accounts for the large increase in time required for this script.

The final script, *Elided Array*, DISPLAYed the same 10-element pointer after setting ARRAY_SIZE and PRINT_SIZE to 5, replacing the default values of 10. The CPU time was cut in half. This reduction came primarily from fewer *writes* due to a smaller PRINT_SIZE. The use of commands to control presentation is effective in improving GDBX performance.

Below are tables of elapsed time and CPU statistics for the various types of display operations.

Elapsed Time (sec)				
Type	DBX Avg	GDBX Avg	Normalized GDBX Avg	% Increase
Startup/Run	7.0	16.5	(9.5)	-
Characters	13.3	26.8	17.3	30.1%
Integers	13.3	27.0	17.5	31.6%
Strings	14.0	28.0	18.5	32.1%
Linked Structure	11.3	33.0	23.5	108.0%
Records	10.3	43.5	34	230.1%
Integer Array	6.5	37.3	27.8	327.7%
Pointer Array	6.0	78.5	69.0	1050.0%
Elided Array	6.0	49	39.5	558.3%

User + System Time (sec)				
Type	DBX Avg	GDBX Avg	Normalized GDBX Avg	% Increase
Startup/Run	1.38	2.4	(1.02)	-
Integers	7.0	8.83	7.81	11.6%
Strings	7.03	9.1	8.08	14.9%
Characters	6.85	8.95	7.93	15.8%
Linked Structs	5.15	11.25	10.23	98.6%
Records	4.68	16.1	14.08	200.9%
Integer Array	3.5	13.55	12.53	258.0%
Pointer Array	3.38	34.25	33.23	883.1%
Elided Array	3.38	20.13	19.11	465.4%

10. Future Work

GDBX succeeded in presenting data structures graphically and in creating a user-tailorable environment. Following are some ideas about how GDBX can be further enhanced and extended.

- A more intelligent layout mechanism for linked structures can be added. The ACROSS/DOWN commands are a beginning in showing how a data structure's layout can be tailored to a particular user or program. Ultimately, a layout algorithm similar to that employed by GRAB [Davis 85], which statically formats directed graphs, can be incorporated into GDBX. Performance would become a concern, however, as layout calculations increase in

complexity.

- The facilities established by GDBX for displaying data structures graphically can be extended to other objects that DBX currently presents textually. For example, the presentation of the call stack or of a structure's definition could be enhanced by using graphic displays.
- Further work on displaying two-dimensional arrays in matrix format or allowing entire sections of an array to be selected, OPENed, or CLOSEd would be useful.
- The graphic presentation abilities of GDBX can be used for applications other than debugging. Programs illustrating algorithms that manipulate data structures, as in Pecan [Reiss 85] or algorithm animation [Brown 85], could be composed using command scripts placed in the *.dbxinit* initialization files.

11. Conclusion

GDBX has shown that graphical displays of data structures can be accomplished in a user-controlled environment. It is a large application built upon the Sun Workstation window manager, a machine and environment that will exert a strong influence on the academic research community in the future.

GDBX was used, in later stages, to debug itself. It proved to be valuable in this self-referential task, aiding in the understanding of complex internal data structures. GDBX has also been well received by the user community at UC Berkeley.

GDBX was implemented by one person over a period of approximately five months. Similar extensions to other debuggers in similar environments would be practical. The value derived by a large user community may make it well worth the effort.

The major strengths of the GDBX project stem from it being integrated fully into a well-known and widely used debugger. GDBX uses the standard debugger command interface to perform its graphical displays and to create debugging environments suited to a particular user or program. GDBX uses the graphics and mouse interface of *SunWindows* to afford the user control over the presentation of data structures, successfully creating a more powerful and effective debugger.

References

[Brown 1985] Marc H. Brown and Robert Sedgewick, "Techniques for Algorithm Animation", IEEE Software, Vol. 2, Number 1, Jan. 1985, pp. 28-39.

[Davis 1985] Michael Davis, "A Layout Algorithm for a Graph Browser", Masters Report, Division of Computer Science, University of California, Berkeley, June, 1985.

[Model 1979] Mitchell I. Model, *Monitoring System Behavior In a Complex Computational Environment*, Palo Alto: Xerox PARC CSL-79-1, January, 1979.

[Myers 1980] Brad A. Myers, *Displaying Data Structures for Interactive Debugging*, Palo Alto: Xerox PARC CSL-80-7, June 1980.

[Reiss 1985] Steven P. Reiss, "PECAN: Program Development Systems That Support Multiple Views", IEEE Transactions on Software Engineering, Vol. SE-11, Number 3, March 1985, pp. 276-285.

[Sun 1983] "Programmer's Reference Manual for SunWindows", Sun Microsystems, Inc., Jan. 7, 1983.

[UNIX 1984] UNIX User's Manual, Reference Guide, "DBX," 4.2 Berkeley Software Distribution, March, 1984.